Master Thesis

EVALUIERUNG VON COUNTERPROPAGATION ARTIFICIAL NEURAL NETWORKS ZUR ERKENNUNG VON ANDROID SCHADSOFTWARE

Masterarbeit zur Erlangung des akademischen Grades

Master of Science

VerfasserIn: Franz Torghele

Vorgelegt am FH-Masterstudiengang MultiMediaTechnology, Fachhochschule Salzburg

Begutachtet durch:
DI Dr. Peter Meerwald MSc (BetreuerIn)
DI Dr. Simon Ginzinger MSc (ZweitgutachterIn)

Puch bei Hallein, 24.11.2014

Eidesstattliche Erklärung

Hiermit versichere ich, Franz Torghele, geboren am **30.07.1987** in **Zell am See**, dass ich die Grundsätze wissenschaftlichen Arbeitens nach bestem Wissen und Gewissen eingehalten habe und die vorliegende Masterarbeit von mir selbstständig verfasst wurde. Zur Erstellung wurden von mir keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ich versichere, dass ich die Masterarbeit weder im In- noch Ausland bisher in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der den BegutachterInnen vorgelegten Arbeit übereinstimmt.

Puch bei Hallein, am **24.11.2014**

Unterschrift

Franz Torghele	1210695004
Vorname Familienname	Personenkennzeichen

Kurzfassung

Smartphone und Tablet Verkäufe nahmen in den vergangenen Jahren stetig zu. Android verfügt mit Abstand über den größten Marktanteil in diesem Segment. Mehr als 1 Milliarde auf Android basierende Smartphones sollen schätzungsweise im Jahr 2017 ausgeliefert werden. Als Folge dieser großen Nachfrage ist die Android Plattform ein lukratives Ziel für EntwicklerInnen von Schadsoftware. Das mobile Betriebssystem sowie der Google Play Store, über welchen Applikationen gewöhnlich bezogen werden, verfügen über Schutzmaßnahmen um schädlichen Anwendungen entgegenzuwirken. Es zeigt sich allerdings, dass diese überwunden werden können und es wichtig ist, die Forschung in diesem Gebiet stetig voranzutreiben. In dieser Arbeit wird die Eignung von Self-Organizing Maps (SOM) beziehungsweise Counterpropagation Artificial Neural Networks (CP-ANN) zur Identifizierung von Schadsoftware unter Android Applikationen evaluiert. Im theoretischen Teil wird auf die Android Plattform und vorhandene Schutzmaßnahmen, auf die statische und dynamische Analyse von Android Applikationen zur Generierung von Feature-Vektoren, sowie auf maschinelles Lernen näher eingegangen. Im praktischen Teil schlägt der Autor ein im Zuge dieser Arbeit entwickeltes Framework namens *AndroSOM* vor, welches mittels statischer und dynamischer Analyse von Android Applikationen, Feature-Vektoren erzeugt. Diese werden anschließend mit Hilfe von SOM beziehungsweise CP-ANN analysiert, um schädliche von legitimen Applikationen zu trennen. Abschließend werden die Ergebnisse diskutiert und ein Ausblick auf weitere Entwicklungen gegeben. Im Zuge dieser Arbeit konnte gezeigt werden, dass sich CP-ANNs gut für die Identifizierung von Android-Schadsoftware eignen.

Schlagwörter:

Android, Schadsoftware, Maschinelles Lernen, Klassifizierung, Self-Organizing Maps, Counterpropagation Artificial Neural Networks

Abstract

Smartphone and tablet sales continuously increased in recent years. Android has by far the largest market share in this segment. More than 1 billion Android-based smartphones and tablets are estimated to be shipped in 2017. As a result of this great success, the Android platform is a lucrative target for developers of malicious software. The mobile operating system as well as the Google Play Store, from which applications usually are obtained, have precautions to counter malicious software. It turns out, however, that this countermeasures can be overcome and it is important to constantly promote research in this area. In this work, the suitability of SOM respectively CP-ANN to identify malicious software among Android applications is evaluated. In the theoretical part of this thesis the Android platform and existing precautions, static and dynamic analysis of Android applications to generate feature vectors as well as self-organizing maps are elaborated. In the practical part, the author proposes a framework called *AndroSOM* which was developed in the course of this master's thesis. It uses static and dynamic code analysis to build strong feature vectors describing Android applications. These are afterwards analyzes using SOM respectively CP-ANN to distinguish legitimate from malicious applications. Finally the results are discussed and an outlook on further developments is given. In the course of this work it was shown that CP-ANN are well suited for the identification of Android malware.

Keywords:

Android, Malware, Machine Learning, Classification, Self-Organizing Maps, Counterpropagation Artificial Neural Networks

Danksagung

Zunächst möchte ich mich bei allen bedanken, die mich in den vergangenen Monaten bei der Erstellung meiner Masterarbeit mit Rat und Tat unterstützt haben. Besonderer Dank gilt dem Studiengang MultiMedia Technology, Herrn Dr. Simon Ginzinger, und allen Teilnehmern des Research Seminars, in dessen Zuge das Thema meiner Masterarbeit stetig weiterentwickelt wurde. Unzählige inhaltliche Diskussionen mit meinen Kollegen gaben ihr zusätzliche Schwerpunkte, die nicht zu unterschätzen sind.

Auch ein besonderer Dank gilt Herrn DI Dr. Peter Meerwald MSc, der die Betreuung meiner Masterarbeit übernommen hat und mich bei inhaltlichen sowie organisatorischen Fragen stets unterstützte.

Ich bin dankbar für die Unterstützung von VirusTotal, einer Tochtergesellschaft von Google, die mir uneingeschränkten Zugriff auf ihre ansonsten kostenpflichtige API gewährt hat, wodurch Teile meiner Arbeit sehr erleichtert wurden.

Nicht zuletzt gebührt besonderer Dank meiner Familie und Freunden: Meinen Eltern für ihre kontinuierliche und bedingungslose Unterstützung, sowie meiner Schwester und Freunden für das Korrekturlesen der Arbeit und für die ständige Motivation, auch oder gerade wenn etwas einmal nicht beim ersten Mal so funktionierte, wie ich es mir vorgestellt hatte.

Inhaltsverzeichnis

1 Einleitung

In den vergangenen Jahren konnte ein rapider Anstieg an Smartphone und Tablet Verkäufen beobachtet werden. Laut Gartner (Gartner 2013), einem bekannten Marktforschungs-Unternehmen, das sich auf die IT Branche spezialisiert hat, stiegen die Smartphone Verkäufe an EndanwenderInnen im zweiten Quartal 2013 um 46,5% an und übertrafen somit das erste Mal die Verkaufszahlen von Feature Phones. Die Android Plattform verfügt mit Abstand über den größten Marktanteil in diesem Segment. Gartner zufolge basierten 79% aller weltweit verkauften Smartphones und Tablets im zweiten Quartal 2013 auf der Android Plattform, gefolgt von iOS mit 14,2%.

Mehr als 1 Milliarde auf Android basierende Smartphones sollen schätzungsweise im Jahr 2017 ausgeliefert werden. Als Folge dieses großen Erfolges ist Android ein lukratives Ziel für EntwicklerInnen von Schadsoftware. Es wird davon ausgegangen, dass jede veröffentlichte schädliche Applikation im Durchschnitt einen sofortigen Gewinn von 10 USD abwirft. (Juniper 2013)

F-Secure (F-Secure 2014) schrieb im „Mobile Thread Report Q1 2014“, dass sich die Anzahl an neu gefundener Schadsoftware seit dem Jahr 2011 vervierfacht hat. Allein im ersten Quartal 2014 wurden 277 neue Varianten identifiziert wovon 99% auf Android abzielten. Davon handelte es sich bei 91% um Software, die ein signifikantes Sicherheitsrisiko für ein Smartphone oder Tablet und die darauf gespeicherten Informationen darstellt. Bei den restlichen 9% handelte es sich um potenziell unerwünschte Applikationen (PUA), welche bei falscher Verwendung die Sicherheit des Smartphones oder Tablets gefährden können.

Die Android Plattform versucht, BenutzerInnen mit einer Reihe von Sicherheitsmechanismen gegen Schadsoftware zu schützen. Jede Applikation wird gewöhnlich in einer eigenen Sandbox ausgeführt, um verschiedene Applikationen voneinander abzuschotten. Dadurch können potenziellen Gefahren durch Programmierfehler minimiert werden und der Schadsoftware wird es erschwert, auf Ressourcen fremder Applikationen zuzugreifen. (Enck, Ongtang und McDaniel 2009)

Der Zugang zu datenschutz- und sicherheitsrelevanten Teilen der Android Programmierschnittstelle (API) wird durch ein Berechtigungs-System gesteuert. Jede Anwendung enthält in ihrer Manifest-Datei Informationen dazu, welche Berechtigungen sie benötigt, um ordnungsgemäß zu funktionieren. BenutzerInnen müssen diese Berechtigungen vor der Installation jeder Anwendung bestätigen. Somit liegt es in der Verantwortung der BenutzerInnen, ob sie einer Anwendung vertrauen oder die Installation anderenfalls abbrechen. Dieses System wird allerdings ineffektiv, wenn Anwendungs-EntwicklerInnen routinemäßig mehr Berechtigungen anfordern, als sie tatsächlich benötigen. (Felt, Chin u. a. 2011)

Dieses Berechtigungs-System stellt einen der wichtigsten Sicherheits-Mechanismen auf Android dar. Es ist jedoch fraglich, wie aufmerksam BenutzerInnen lesen, welche Berechtigungen von einer Anwendung tatsächlich angefordert werden. Einer Studie zufolge (Felt u. a. 2012) beachteten nur 17% der Teilnehmer während der Installation von Applikationen die angeforderten Berechtigungen und nur 3% konnten drei Fragen zu diesen richtig beantworten. Aktuelle Schadsoftware-Beispiele zeigen auch, dass über 100.000 Android

BenutzerInnen „Super Mario Bros“ oder „GTA 3 – Moscow City“ auf ihren Endgeräten installiert hatten, obwohl sie vor der Installation gefragt wurden, ob sie „*Services that cost you money: send SMS messages*“ erlauben wollen. (Zdnet 2012)

1.1 Relevanz

Durch den enormen Zuwachs an Android Schadsoftware in den vergangenen Jahren müssen Sicherheits-ExpertInnen und HerstellerInnen von Antivirus-Lösungen eine immer größer werdende Anzahl schädlicher Applikationen analysieren, um deren Wirkungsweise zu verstehen und Gegenmaßnahmen zu entwickeln.

Solch eine Analyse geschieht traditionell durch das De-kompilieren von Applikationen und mit Hilfe von Debuggern. Je nach Erfahrung des Analysten oder der Analystin kann diese Arbeit sehr zeitaufwendig und fehleranfällig sein. Darum gibt es seit Jahren das Bestreben, die Analyse von Android Anwendungen zu automatisieren. In Kapitel 3 wird dazu näher auf verschiedene Herangehensweisen und Methoden eingegangen.

Da sich Schadsoftware ständig weiterentwickelt, darf auch deren Bekämpfung keinem Stillstand unterliegen. Es ist wichtig immer neue Wege zu erforschen, die zu einer möglichst guten Erkennungsrate führen. In dieser Arbeit soll dafür die Eignung von Self-Organizing Maps (SOM) beziehungsweise von Counterpropagation Artificial Neural Networks (CP-ANN), unter Verwendung von Feature-Vektoren aus statischer und dynamischer Analyse, analysiert werden.

In (Barrera u. a. 2010) wurden SOMs bereits erfolgreich zur Analyse von Applikations-Berechtigungen, einem der grundlegenden Sicherheitsmechanismen unter Android, eingesetzt. Diese Arbeit unterscheidet sich dahingehend, dass nicht Berechtigungen alleine als Grundlage für die SOM und das CP-ANN herangezogen werden, sondern Daten, die aus statischer und dynamischer Analyse stammen.

1.2 Methode

Software kann grundsätzlich auf zwei verschiedene Arten analysiert werden - statisch oder dynamisch. Bei der statischen Analyse wird der Quellcode einer Anwendung untersucht. Bei der dynamischen Analyse wird das Verhalten einer Anwendung überwacht, während sie in einer kontrollierten Umgebung ausgeführt wird.

In dieser Arbeit werden statische sowie dynamische Analyse eingesetzt, um ein möglichst gutes Gesamtbild der zu analysierenden Applikationen zu erhalten. Dazu wurde ein Framework namens „AndroSOM Feature Miner“ entwickelt. Besonderes Augenmerk lag hierbei darauf, die dynamische Analyse möglichst performant zu gestalten. Anschließend wurden die gesammelten Daten mit Hilfe von SOM beziehungsweise CP-ANN analysiert und versucht, schädliche von legitimen Applikationen zu trennen.

1.3 Forschungsfrage

Im Zuge dieser Arbeit soll folgende Fragestellung beantwortet werden:

Wie zuverlässig lassen sich schädliche von legitimen Android Applikationen mit Hilfe von Counterpropagation Artificial Neural Networks (CP-ANNs), unter Verwendung von Feature-Vektoren aus statischer und dynamischer Analyse, unterscheiden?

1.4 Aufbau der Arbeit

Der Hauptteil dieser Arbeit ist in sechs Kapitel aufgeteilt:

- In Kapitel 2 wird auf Grundlagen zu Android eingegangen, die für das Verständnis dieser Arbeit von Nöten sind. Darunter befindet sich ein Überblick über die System-Architektur, die Komponenten und die Funktionsweise von Android Applikationen sowie vorhandene Sicherheitsmechanismen samt möglicher Angriffspunkte.
- In Kapitel 3 wird auf verschiedene Arten von Schadsoftware, sowie auf deren Abwehr und Analyse eingegangen. Hier werden vor allem bestehende Arbeiten vorgestellt. Anschließend folgt eine Einführung in Maschinelles Lernen (ML), Artificial Neural Networks (ANN), SOM und CP-ANN.
- In Kapitel 4 wird der „AndroSOM Feature Miner“ vorgestellt und näher auf dessen Implementierung und Design-Entscheidungen eingegangen. Außerdem wird beschrieben, wie aus den gesammelten Rohdaten Feature-Vektoren entstehen.
- In Kapitel 5 werden die Messmethoden beschrieben und auf die Resultate näher eingegangen.
- In Kapitel 6 werden die Ergebnisse diskutiert, die Forschungsfrage beantwortet und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2 Android Grundlagen

In diesem Kapitel erfahren Sie Grundlagen zu Android, die für das weitere Verständnis dieser Arbeit von Nöten sind. Es wird näher auf die Android Plattform und dessen System-Architektur, sowie auf die verschiedenen Komponenten in Android Applikationen eingegangen. Abschließend wird ein Unterkapitel den vorhandenen Sicherheitsmechanismen unter Android und bestehenden Angriffspunkten gewidmet.

2.1 Das Android Betriebssystem

Bei Android handelt es sich um ein Betriebssystem, das vorwiegend für mobile Endgeräte, wie etwa Smartphones oder Tablets, entwickelt wird. Das Betriebssystem dient hierbei als Schnittstelle zwischen Hardwarekomponenten und den darauf laufenden Anwendungen.

Android Anwendungen werden meist in Java geschrieben und in einer Virtuellen Maschine (VM) ausgeführt. Diese speziell für Android entwickelte VM nennt sich Dalvik und ist eine der Kern-Komponenten des Betriebssystems. In ihr werden alle Applikationen und einige System-Services ausgeführt (Google 2014g). Mit Android 4.4 (KitKat) wurde eine neue experimentelle VM namens Android Runtime (ART) eingeführt, welche die Dalvik VM voraussichtlich in kommenden Versionen des mobilen Betriebssystems, beginnend mit Android 5.0, ersetzen wird (Google 2014h).

Version	Name	API-Levels	Release Zeitraum	Anzahl
1.0	Base	1-2	Sep. 2008 - Feb. 2009	2
1.5	Cupcake	3	Apr. 2009	1
1.6	Donut	4	Sep. 2009	1
2.0-2.1	Eclair	5-7	Okt. 2009 - Jan. 2010	3
2.2	Froyo	8	Mai 2010 - Jan. 2011	3
2.3	Gingerbread	9-10	Dez. 2010 - Sep. 2011	8
3.x	Honeycomb	11-13	Feb. 2011 - Sep. 2011	4
4.0	Ice Cream Sandwich	14-15	Okt. 2011 - Feb. 2012	5
4.1-4.3	Jelly Bean	16-18	Jun. 2012 - Okt. 2013	8
4.4	KitKat	19	Okt. 2013 - Jun. 2014	5
4.4W	Wear	20	Jun. 2014	1

Tabelle 1: Überblick Android Versionen
Quelle: https://source.android.com/source/build-numbers.html

Das mobile Betriebssystem wurde ursprünglich von Android Inc. entwickelt und nach dem Verkauf an Google im Jahr 2007 als Android Open Source Project (AOSP) veröffentlicht. Von diesem Zeitpunkt an wurde Android von der Open Handset Alliance (OHA), einem Zusammenschluss von 78 Firmen, weiterentwickelt. Der Großteil des Android Quellcodes wird unter der „Apache Software License, Version 2.0“ zur Verfügung gestellt und kann von einem zentralen Archiv frei bezogen und auch modifiziert werden (Google 2014j).

Wie in Tabelle 1 zu sehen ist, schreitet die Entwicklung der Android Plattform schnell voran. In kurzen Abständen erscheinen neue Major-Releases. Dieser Umstand führt dazu, dass Informationen über Android, besonders in Büchern oder Artikeln, sehr schnell veralten. Als aktuelle Quellen können oftmals nur die Software Development Kit (SDK) Dokumentation oder Blogs dienen.

2.2 Überblick System-Architektur

Wie in Abbildung 1 zu sehen ist, kann das Android Grundsystem grob in vier Schichten unterteilt werden: dem Linux Kernel samt grundlegenden Tools, den System-Bibliotheken, der Android Runtime, dem Applikationsframework und der darauf aufbauenden Applikationsschicht.

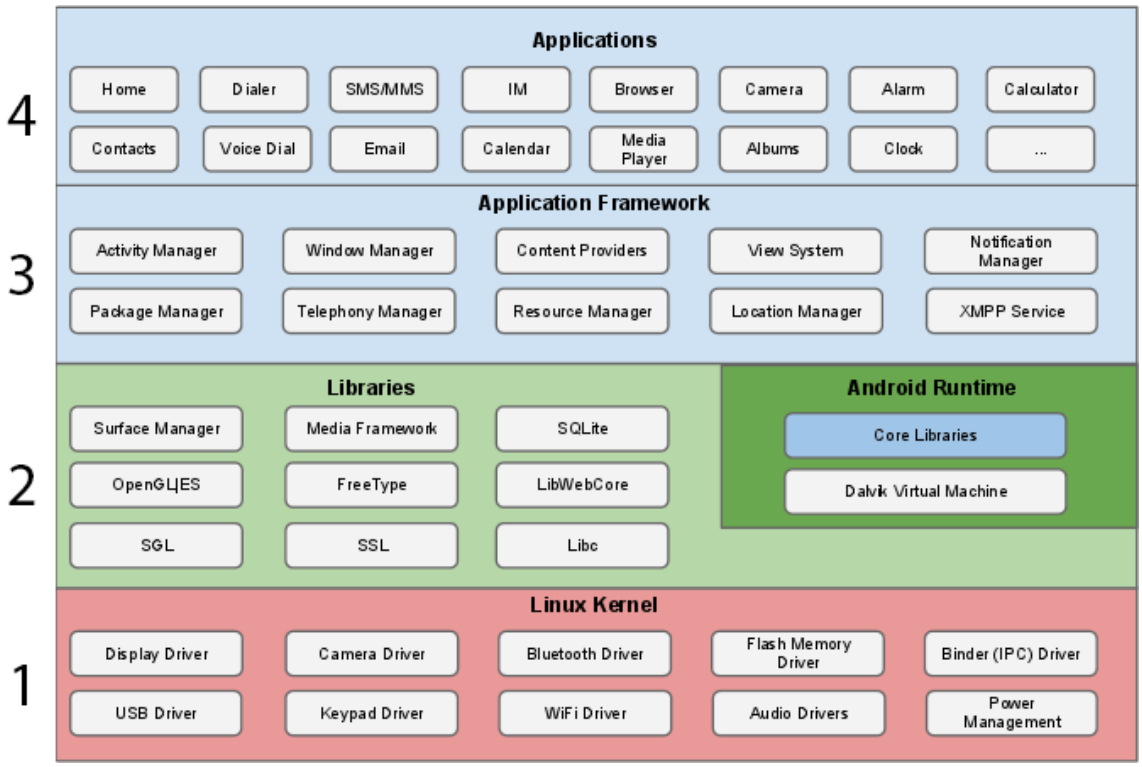


Abbildung 1: Android System-Architektur (Google 2014e)

Die verschiedenen Schichten setzen sich aus mehreren Komponenten zusammen, auf welche in den folgenden Kapiteln genauer eingegangen wird.

Jede Schicht stellt dabei den darüber liegenden Schichten Funktionalität zur Verfügung. Die Kommunikation zwischen den verschiedenen Komponenten und Schichten erfolgt meist über einen Android spezifischen Inter-process Communication (IPC) Mechanismus namens Binder.

2.2.1 Linux Kernel

Die unterste Schicht bildet der Linux Kernel. Das gesamte Android Betriebssystem baut auf dem Linux 2.6 Kernel auf, welcher einigen architektonischen Änderungen unterzogen wurde. Ein Beispiel hierfür ist die IPC-Schnittstelle Binder, welche als Kernel-Modul implementiert ist. Der Kernel ist für die Interaktion mit der Hardware verantwortlich und beinhaltet alle wichtigen Treiber, beispielsweise für das Global Positioning System (GPS) Modul, Kamera, Audio und Bluetooth. Der Kernel dient somit als Abstraktions-Schicht zwischen der Hardware der Endgeräte und den darüber liegenden Software-Schichten. Von ihm wird die gesamte Kern-Funktionalität abgedeckt, wie etwa die Speicher-, Prozess-, Netzwerk- und Energieverwaltung. Die Verwendung des bewährten Linux Kernels macht es außerdem einfacher, Android auf verschiedene Endgeräte zu portieren (App-Market 2012).

2.2.2 System-Bibliotheken

Da Android auch auf mobilen Endgeräten mit wenig Hauptspeicher und einer langsamen Central Processing Unit (CPU) laufen muss, werden System-Bibliotheken für CPU- und Graphics Processing Unit (GPU)-intensive Rechenaufgaben zu stark optimiertem, nativem Code kompiliert (App-Market 2012).

Einige der wichtigsten nativen Bibliotheken sind folgende:

- **Surface Manager:** Der Surface Manager dient der Anzeigeverwaltung. Er entscheidet, welche Applikationen zu welcher Zeit ihre Inhalte auf dem Bildschirm des Endgerätes anzeigen dürfen.
- **OpenGL|ES:** OpenGL|ES kann für die Darstellung von 3D Inhalten verwendet werden. Verfügt das Endgerät über Hardwarebeschleunigung, kann diese genutzt werden, was auch aufwendige 3D Inhalte, wie sie etwa in Spielen anzutreffen sind, ermöglicht.
- **SGL:** Der Großteil der Applikationen verwendet allerdings eine Android Bibliothek namens „Scalable Graphics Library“, welche auf das Darstellen von 2D Inhalten ausgelegt ist.
- **Media Framework:** Das Media Framework beinhaltet verschiedene Codecs zum performanten Aufnehmen und Abspielen von Medien-Inhalten wie zum Beispiel von Videos.
- **SQLite:** SQLite steht unter Android Applikationen als Datenbank zur Verfügung[1].

Android unterstützt momentan drei Prozessor-Architekturen, für die diese Bibliotheken zu nativem Code kompiliert werden können: ARM, x86 und MIPS (Google 2014i).

1. http://developer.android.com/reference/android/database/sqlite/package-summary.html

2.2.3 Android Runtime

Bei der Android Runtime handelt es sich um diejenige Komponente, die das Betriebssystem am meisten von einer herkömmlichen mobilen Linux Implementation unterscheidet. Die Runtime beinhaltet die Dalvik VM sowie grundlegende Android Java Bibliotheken, die das Programmieren von Android Applikationen erst ermöglichen (Meier 2012, S. 15).

Android Bibliotheken

Bei diesen Bibliotheken handelt es sich nicht um herkömmliche JavaSE oder JavaME Bibliotheken. Viel mehr handelt es sich um stark optimierte Nach-Implementierungen der bewährten JavaSE Bibliotheken, gepaart mit Android spezifischer Funktionalität (App-Market 2012).

Dalvik VM

Android Applikationen und das darunter liegende Applikationsframework werden fast zur Gänze in Java geschrieben. Anstatt eine herkömmliche Java Virtual Machine (JVM), wie zum Beispiel die der JavaME zu verwenden, wurde für Android eine eigene VM namens Dalvik entwickelt. Diese virtuelle Maschine ist auch nicht zur JVM kompatibel, da sie auf leistungsschwache mobile Systeme optimiert wurde. Mobile Systeme verfügen oftmals nur über wenig Hauptspeicher, eine langsame CPU und anders als herkömmliche Systeme über keine Auslagerungs-Partition, um den ohnehin kleinen Hauptspeicher auszugleichen (Bornstein 2008).

Im Gegensatz zur JVM werden in der Dalvik VM die .class-Dateien nicht direkt ausgeführt. Die Dalvik VM verfügt über ihr eigenes Bytecode Format, welches speziell auf mobile Endgeräte abgestimmt wurde. Dieser Dalvik-Bytecode ist kleiner als normaler Java-Bytecode und bietet bessere Effizienz in Umgebungen mit geringen Ressourcen. Dadurch wird es ermöglicht, mehrere Instanzen der virtuellen Maschine effizient nebeneinander auf mobilen Endgeräten auszuführen (Meier 2012, S. 16) .

Der Bytecode wird der virtuellen Maschine hierfür in Form von Dalvik Executable Format (DEX)-Dateien, welche zu Dalvik-Bytecode kompilierte .class-Dateinen enthalten, übergeben. Die Dalvik VM bietet außerdem ein hoch effizientes Speichermanagement, bei dem ein Prozess namens Zygote (siehe Kapitel 2.3.6) eine wichtige Rolle spielt, sowie Multithreading-Unterstützung (App-Market 2012).

ART

Mit Android 4.4 (KitKat) wurde eine neue experimentelle VM namens ART eingeführt, welche die Dalvik VM voraussichtlich in kommenden Versionen des mobilen Betriebssystems ersetzen wird (Google 2014h).

Einige der Vorteile von ART gegenüber der Dalvik VM sind folgende:

- **Ahead Of Time (AOT) Kompilierung:** Android Anwendungen werden in Zukunft zur Installationszeit aus DEX-Dateien zu nativem Code kompiliert. Dadurch soll die Performance zur Laufzeit verbessert werden.
- **Verbesserte Garbage Collection (GC):** Eine ineffektive GC kann die Reaktionsfähigkeit von Anwendungen negativ beeinflussen. Dem soll durch eine verbesserte GC entgegengewirkt werden.
- **Verbessertes Debuggen von Anwendungen:** EntwicklerInnen soll das Suchen von Fehlern in ihren Applikationen erleichtert werden.

2.2.4 Applikationsframework

Hierbei handelt es sich um die Komponenten, mit denen Applikationen direkt interagieren. Das Applikationsframework dient außerdem als weitere Abstraktionsschicht für den Zugriff auf Hardware-Komponenten sowie Ressourcen (Meier 2012, S. 15).

Das Applikationsframework besteht unter anderem aus folgenden Bestandteilen:

- **Activity Manager:** Der Activity Manager regelt den Lebenszyklus von Applikationen (siehe Kapitel 2.3.6).
- **Package Manager:** Der Package Manager ist für das Installieren, Deinstallieren und Updaten von Applikationen verantwortlich. Außerdem kann er Informationen zu allen auf dem Endgerät installierten Anwendungen liefern.
- **Window Manager:** Der Window Manager verwaltet, welche Fenster am Bildschirm wie angezeigt werden. Unter anderem führt er auch automatische Übergänge von Fenstern, sowie Animationen beim Öffnen oder Schließen von Applikationen und beim Drehen des Bildschirms durch.
- **Telephony Manager:** Der Telephony Manager ist für eingehende und ausgehende Anrufe verantwortlich.
- **Content Providers:** Content Provider können für den Austausch von Daten zwischen verschiedenen Applikationen verwendet werden, mehr dazu in Kapitel 2.3.1.
- **Resource Manager:** Der Resource Manager verwaltet alle Ressourcen, wie zum Beispiel Bilder oder Extensible Markup Language (XML)-Dateien, auf die Applikationen Zugriff haben.
- **Location Manager:** Der Location Manager stellt Positionsdaten zur Verfügung. Diese kann er entweder durch GPS, WLAN oder das Mobilfunknetz beziehen.

2.2.5 Applikationsschicht

Die Applikationsschicht ist die oberste Schicht in der Android Architektur. Für diese können EntwicklerInnen Applikationen schreiben. Es gibt einige Systemapplikationen, die mit jedem Android System ausgeliefert werden, beispielsweise eine Anwendung für den Startbildschirm, das Telefon, das Schreiben und Empfangen von SMS-Nachrichten, die Verwaltung von Kontakten und Terminen, das Surfen im Internet, das Schreiben und Empfangen von Emails, das Erstellen und Betrachten von Bildern, die Anzeige einer Uhr oder das Stellen eines Weckers.

EntwicklerInnen ist es unter Android sogar erlaubt, jede dieser Systemapplikationen durch eigene Implementierungen zu ersetzen. Dadurch sind sie nicht limitiert, was die Anpassung von Android angeht (App-Market 2012).

Neben den Systemapplikationen existiert eine große Anzahl von Applikationen von Drittanbietern. Diese können beispielsweise aus dem offiziellen Android Play Store oder alternativen Marktplätzen, wie Amazon, bezogen werden. Auf den Aufbau von Android Applikationen wird in Kapitel 2.3 näher eingegangen.

2.2.6 Binder

Binder wurde ursprünglich unter dem Namen OpenBinder von Be Inc. entwickelt und von Android übernommen. Aus Lizenzgründen wurde OpenBinder für Android komplett neu implementiert.

Bei Binder handelt es sich um einen leichtgewichtigen, hoch performanten Remote Procedure Call (RPC) Mechanismus, der sowohl innerhalb von Prozessen als auch prozessübergreifend eingesetzt werden kann (Google 2014e).

Binder wurde als Linux Kernel-Modul implementiert, über welches Prozesse kommunizieren können. Dieses wird an Stelle von herkömmlichen Linux IPC Mechanismen eingesetzt, um IPC Operationen effizient als „Thread Migration“ zu modellieren. Thread Migration bedeutet, dass der Thread, der die IPC veranlasst, in den Ziel Prozess wechselt, dort Code ausführt und mit dem Ergebnis anschließend zurück in den ursprünglichen Prozess wechselt.

Aus Gründen der Sicherheit wurde der Binder IPC Mechanismus von Android allerdings ohne Thread Migration implementiert. Stattdessen verwaltet Binder einen Pool an Threads in jedem Prozess, die dafür genutzt werden, hereinkommende IPCs und lokale Anfragen abzuarbeiten. Das Kernel Modul emuliert im Fall von Android nur Thread Migration und stellt sicher, dass IPCs von ihrem ursprünglichen Thread entgegengenommen werden, wenn ein IPC in den initiierenden Prozess zurückkehrt.

Das Binder Kernel Modul ist auch dafür verantwortlich, Objekt-Referenzen über Prozessgrenzen hinweg zu verfolgen. Entfernte Objekt-Referenzen werden dafür auf das jeweilige echte Objekt abgebildet. Dadurch wird sichergestellt, dass keine Objekte zerstört werden, solang ein entfernter Prozess eine Referenz auf diese hält (PalmSource 2005).

Intent

Ein Intent ist eine Nachricht, die über die Binder IPC Schnittstelle geschickt wird. Dabei handelt es sich um eine abstrakte Repräsentation einer Operation, die ausgeführt werden soll. Abstrakt deshalb, weil nicht definiert sein muss, durch welche Komponente diese Operation ausgeführt werden soll. Ein Intent verfügt zumindest über ein `Action` und ein `Data` Feld, kann aber auch noch zusätzliche Informationen enthalten (Google 2014b). Tabelle 2 zeigt einige Beispiele, wie ein Intent aussehen kann.

Action	Data
ACTION_VIEW	content://contacts/people/
ACTION_DEAL	content://contacts/people/1
ACTION_EDIT	content://contacts/people/1
ACTION_DIAL	tel:123

Tabelle 2: Android Intents (Google 2014b)

Wenn ein Intent an eine bestimmte Komponente geschickt wird, handelt es sich um einen expliziten Intent. Wenn allerdings im Falle eines impliziten Intents keine Komponente definiert wurde, entscheidet das System, mit welcher Komponente die Operation ausgeführt wird oder bietet BenutzerInnen die Möglichkeit, selbst aus mehreren Alternativen zu wählen (Enck, Ongtang und McDaniel 2009).

2.3 Android Applikationen

Android Applikationen werden großteils in Java geschrieben, können allerdings durch das Android Native Development Kit (NDK) auch native Komponenten (siehe Kapitel 2.3.5) enthalten. Mit Hilfe der Android SDK Werkzeuge wird der Quellcode kompiliert und zusammen mit allen benötigten Ressourcen zu einem Android Package (APK) zusammengefasst. Dabei handelt es sich um eine Archivdatei mit der Endung `.apk`, welche später auf Android Endgeräten installiert werden kann. (Google 2014f)

2.3.1 Komponenten

Eine Android Applikation besteht aus verschiedenen Komponenten. Jede Komponente kann dabei einen Einstiegspunkt in die Applikation darstellen. Es gibt vier verschiedene Arten von Komponenten. Jede davon dient einem bestimmten Zweck und verfügt über einen eigenen Lebenszyklus.

Aktivitäten

Eine „Activity“ oder auch Aktivität repräsentiert eine einzelne Benutzeroberfläche. Zum Beispiel kann eine Kalender-Applikation eine Aktivität mit einer Wochenübersicht, eine

weitere Aktivität zum Eintragen eines Termins und eine Aktivität zum Anzeigen desselben besitzen. Obwohl diese verschiedenen Aktivitäten zusammenarbeiten, um für nahtlose Übergänge in der Kalender-Applikation zu sorgen, ist jede unabhängig von den anderen. Es ist sogar möglich, dass eine externe Applikation eine bestimmte Aktivität der Kalender-Applikation startet, sofern die Kalender-Applikation dass erlaubt. Beispielsweise könnte eine externe Geburtstags-Applikation die Aktivität zum Eintragen eines Termins der Kalender-Applikation verwenden. (Google 2014f)

Auf nähere Details zum Lebenszyklus einer Aktivität wird in Kapitel 2.3.6 eingegangen.

Dienste

Bei einem „Service“ oder auch Dienst handelt es sich um eine Komponente, die im Hintergrund ausgeführt wird. Dienste werden vor allem für lang andauernde Operationen eingesetzt oder etwa um rechenaufwendige Aufgaben für Aktivitäten zu lösen, ohne deren GUI zu blockieren. Dienste verfügen über keine eigene GUI. Als Beispiel könnte ein Dienst Musik im Hintergrund abspielen oder Daten von einem externen Server herunterladen, während sich der oder die BenutzerIn gerade in einer anderen Applikation befindet. Dienste können von einer Aktivität oder durch einen Broadcast Receiver gestartet werden.

Ein Dienst ist selbst für seinen Lebenszyklus verantwortlich und wird normalerweise nicht automatisch vom System beendet (Google 2014f).

Content Providers

Content Provider sind für die Verwaltung der Daten einer Applikation verantwortlich. EntwicklerInnen können Daten auf dem Dateisystem, in einer SQLite Datenbank, im Internet oder auf jeglichem anderen persistenten Speicherort ablegen, auf den sie Zugriff haben. Durch einen Content Provider wird es externen Applikationen ermöglicht, auf diese Daten zuzugreifen oder diese sogar zu ändern, sofern sie über die nötigen Berechtigungen verfügen. Das Android Betriebssystem stellt zum Beispiel einen Content Provider zur Verfügung, der Zugriff auf Kontaktdaten bietet. Jede Anwendung, die über die nötigen Berechtigungen verfügt, kann somit Kontaktdaten abfragen oder auch Informationen über einen bestimmten Kontakt bearbeiten (Google 2014f).

Broadcast Receivers

Ein Broadcast Receiver ist eine Komponente, die auf systemweite Broadcast-Mitteilungen reagiert. Viele Broadcasts stammen direkt von Android, beispielsweise die Nachricht, dass der Bildschirm ausgeschaltet wurde, die Batterie schwach ist, ein Anruf hereinkommt oder der Bootvorgang abgeschlossen wurde. Auch Applikationen können Broadcasts aussenden, beispielsweise um anderen Applikationen mitzuteilen, dass bestimmte Daten von nun an zur weiteren Nutzung bereitstehen. So wie zuvor bereits Dienste verfügen auch Broadcast Receiver über keine eigene GUI, sie können allerdings eine Mitteilung in der Android Statusbar erstellen, um auf ein bestimmtes Ereignis hinzuweisen. Meistens verrichtet ein

Broadcast Receiver nur sehr wenig oder gar keine Arbeit, sondern startet eine andere Komponente, meist einen Dienst, welcher diese Arbeit anschießend, basierend auf dem empfangenen Ereignis, verrichtet (Google 2014f).

Wie eine andere Komponente gestartet werden kann, wird im folgenden Unterkapitel beschrieben.

2.3.2 Aktivierung von externen Komponenten

Ein einzigartiger Aspekt des Android Betriebssystems ist die Tatsache, dass jede Applikation Komponenten einer anderen starten und benutzen kann. Wenn EntwicklerInnen zum Beispiel wollen, dass BenutzerInnen mit der Kamera des Endgerätes Videos aufnehmen, kann diese Funktionalität von einer externen Applikation bereitgestellt werden. Das aufgenommene Video wird anschließend an die eigene Applikation zur Weiterverarbeitung übergeben. Den EntwicklerInnen bleibt somit die Implementierung einer eigenen Aktivität zum Aufnehmen von Videos erspart. Dafür muss nicht einmal auf den Programmcode der externen Kamera-Applikation zugegriffen, sondern einfach nur deren Aktivität zum Aufnehmen von Videos gestartet werden. Für BenutzerInnen der Applikation erscheint es so, als ob die Kamera-Funktionalität Teil dieser wäre.

Wenn eine Komponente gestartet werden soll, wird zuerst ein neuer Prozess, beziehungsweise eine neue VM, erstellt (wenn diese nicht bereits im Hintergrund läuft) und die Klassen, welche für die Komponente benötigt werden, instantiiert. Wenn wie im obigen Beispiel die Kamera-Aktivität gestartet wird, wird diese nicht in der VM der eigenen Anwendung ausgeführt. So kann die Isolation der Anwendungen aufrecht erhalten bleiben. Da es sich um verschiedene VMs beziehungsweise Prozesse handelt, können Android Applikationen die Komponenten von anderen Anwendungen auch nicht direkt starten (Google 2014f).

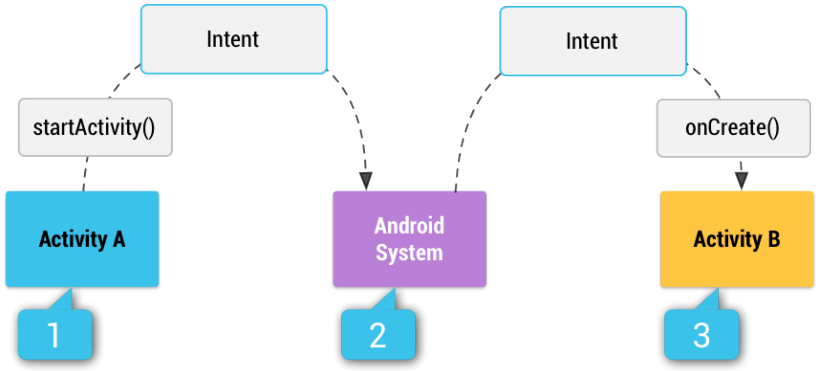


Abbildung 2: Start einer externen Aktivität (Google 2014c)

Wie in Abbildung 2 zu sehen ist, geschieht der Start von externen Komponenten, oder in diesem Fall einer Aktivität, durch einen Intent (siehe Kapitel 2.2.6). In Schritt 1 wird ein Intent, mit dem Wunsch eine bestimmte Aktivität zu starten, an `startActivity()`

übergeben. In Schritt 2 sucht das Android System unter allen installierten Anwendungen, ob eine entsprechende Aktivität vorhanden ist. Wurde diese gefunden, kann sie in Schritt 3 vom Android System gestartet werden (Google 2014c).

Welche Komponenten eine Applikation enthält, wird in ihrem Manifest deklariert.

2.3.3 Android Manifest

Bevor das Android System eine Applikations-Komponente starten kann, muss es wissen, dass diese Komponente existiert. Aus diesem Grund verfügt jede Android Applikation in ihrem Stammverzeichnis über eine Datei namens „AndroidManifest.xml“. In dieser Manifest-Datei müssen alle Komponenten deklariert sein (Google 2014f).

Zusätzlich zur Deklaration der Komponenten, enthält das Android Manifest weitere wichtige Informationen, unter anderem folgende:

- Alle Berechtigungen, wie zum Beispiel Zugriff auf das Internet oder das Lesen von Kontaktdaten, die von der Applikation benötigt werden.
- Das minimale API Level (siehe Tabelle 1), welches von der Applikation vorausgesetzt wird.
- Deklaration aller Hardware-Features, wie zum Beispiel Kamera oder GPS Modul, auf die von der Applikation zugegriffen wird.
- Deklaration von zusätzlichen Bibliotheken, die nicht bereits im Android Framework enthalten sind, wie zum Beispiel die „Google Maps Bibliothek“.

Listing 1 zeigt den Inhalt der Android Manifest Datei einer Beispielapplikation. Darin werden das Paket (`at.torghele.camera`), die Version der Applikation, das minimale API-Level sowie benötigte Berechtigungen (`android.permission.CAMERA`) und Features (`android.hardware.camera` und `android.hardware.camera.autofocus`) aufgelistet. Außerdem wurde eine Aktivitäts-Komponente deklariert.

2.3.4 Ressourcen

Android Anwendungen bestehen nicht nur aus Programmcode, sondern beinhalten auch andere Ressourcen, beispielsweise Bilder oder XML-Dateien. Unter Android werden zum Beispiel Animationen, Menüs, Farben und das Layout von GUI-Elementen in XML-Dateien gespeichert (Google 2014f). Die Vorteile sind folgende:

- Eigenschaften einer Applikation, wie zum Beispiel das Erscheinungsbild, können an einer zentralen Stelle, und nicht im Code selbst, modifiziert werden.
- Es ist einfach, mehrere alternative Ressourcen, zum Beispiel Bilder mit verschiedenen Auflösungen für verschiedene Endgeräte, bereitzustellen.

Listing 1: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="at.torghele.camera"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-sdk android:minSdkVersion="8" />
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-feature android:name="android.hardware.camera" />
  <uses-feature android:name="android.hardware.camera.autofocus" />
  <application android:label="@string/app_name" >
    <activity android:name="CameraActivity"
              android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

2.3.5 Nativer Code

Seit Juni 2009 bietet Google mit dem NDK die Möglichkeit, Android Applikationen nicht nur in Java, sondern zum Teil auch in C/C++ zu programmieren. Diese Vorgehensweise kann für manche Applikationen hilfreich sein, da auf existierende Bibliotheken, die in diesen Sprachen implementiert wurden, zurückgegriffen werden kann. Gewöhnliche Android Applikationen profitieren allerdings nicht oder nur selten von der Verwendung von nativen Bibliotheken, da deren Performance-Vorteil meist nicht im Verhältnis mit der zusätzlichen Komplexität der Applikationen steht. Typische Kandidaten für die Verwendung des NDK sind CPU- und GPU-intensive Anwendungsbereiche, wie beispielsweise Spiele-Engines, Signalverarbeitung oder Physik-Simulationen (Google 2014d) .

Der Performance Unterschied von nativem C/C++ Code zu in der Dalvik-VM interpretiertem Code wurde in (Lee und Jeon 2010) untersucht. Besonders hervorzuheben ist hierbei eine starke Performance-Steigerung im Hinblick auf Speicher-Operationen und Integer Kalkulationen.

Kompatibilität

Native Bibliotheken, welche mit Hilfe des Android NDKs erstellt wurden, können auf den Android Plattformen ab einem bestimmten API-Level verwendet werden. Um welches API-Level es sich dabei handelt, hängt von der CPU Architektur des Endgerätes ab.

CPU Architektur	kompatible Android Plattformen
ARM, ARM-NEON	Android 1.5 (API-Level 3) und höher
x86	Android 2.3 (API-Level 9) und höher
MIPS	Android 2.3 (API-Level 9) und höher

Tabelle 3: Android NDK Kompatibilität

Wie in Tabelle 3 zu sehen ist, genügt bei ARM-basierten Systemen das API-Level 3 beziehungsweise Android 1.5 oder eine neuere Version. Ab API-Level 2.3 werden auch x86 und MIPS unterstützt.

Um native Bibliotheken zu implementieren, die auf die OpenGL|ES APIs zugreifen, wird API-Level 4 für OpenGL|ES 1.1 beziehungsweise API-Level 5 für OpenGL|ES 2.0 vorausgesetzt.

2.3.6 Start einer Applikation

Jede Android Applikation läuft standardmäßig in ihrem eigenen Linux Prozess mit nur einem Thread. Dieser Haupt-Thread besitzt eine Instanz von `android.os.Looper`, in der alle Aufgaben nacheinander abgearbeitet und an die entsprechenden Funktionen weitergeleitet werden.

Ein solcher Anwendungs-Prozess wird immer dann gestartet, wenn er benötigt wird und nicht bereits im Hintergrund läuft. Dies kann der Fall sein, wenn die Applikation von einem Benutzer oder einer Benutzerin gestartet wird oder der Start einer ihrer Komponenten von einem Dienst, einem Broadcast Receiver oder einer Aktivität angefordert wird. Prozesse werden gewöhnlich vom System so lange am Leben erhalten wie möglich, auch wenn keine Interaktion durch BenutzerInnen stattfindet. Sie werden erst vom System beendet, wenn es zu Ressourcen-Engpässen kommt. Dabei werden diejenigen Prozesse als erstes beendet, die am längsten inaktiv beziehungsweise im Hintergrund waren. Auf diese Art wird das aufwendige Starten und Beenden der VMs, in denen normalerweise jeweils nur eine Anwendung läuft, minimiert (Karandikar 2010).

Um das Starten der Dalvik VM für jede einzelne Applikation zu beschleunigen und um Ressourcen zu sparen, gibt es unter Android einen besonderen Prozess namens Zygote.

Der Zygote Prozess

Dieser Prozess wird sehr früh während des Boot-Vorganges des Android Systems gestartet. Es handelt sich dabei um die erste Instanz der Dalvik VM, in der bereits alle Applikationsframework-Klassen und andere System-Bibliotheken geladen werden. Danach hört der Zygote Prozess auf ein Socket-Interface, um bei Bedarf neue VMs für andere Applikationen zu starten. Wird eine neue VM vom System angefordert, forkt sich der Zygote Prozess selbst und stellt somit eine vor-initialisierte Instanz der Dalvik VM samt Applikationsframework und benötigter Bibliotheken zum Ausführen einer Android Applikation zur Verfügung (Karandikar 2010).

Beim Aufruf von `fork()` erstellt der aktuelle Prozess eine Kopie von sich selbst. Da dies allerdings mittels *Copy-on-Write* geschieht, muss vorerst kein physischer Speicher dupliziert werden. Stattdessen können die virtuellen Speicherseiten in beiden Prozessen auf dieselben physischen Speicherseiten zeigen. Erst wenn eine dieser Speicherseiten verändert wird, muss diese dupliziert werden (Bach 1986).

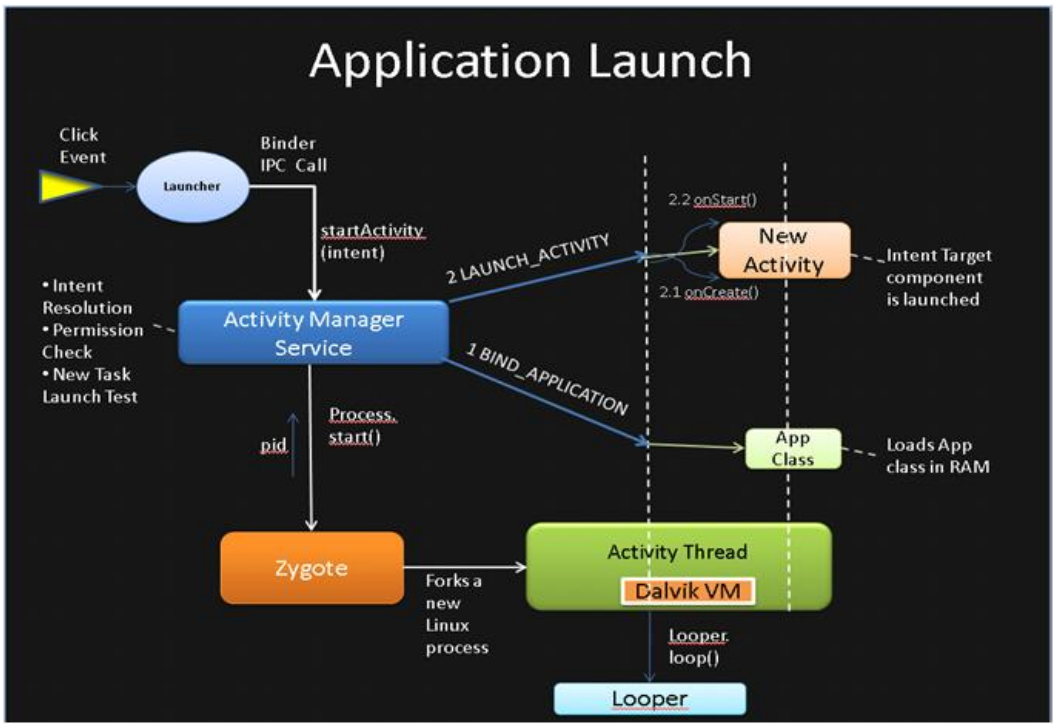


Abbildung 3: Start einer Android Applikation (Karandikar 2010)

Abbildung 3 zeigt den Start einer Android Applikation. Nach einem Klick-Event wird zuerst ein `Intent` an den `Activity Manager Service` geschickt. Dieser sammelt anschließend Informationen über das Ziel des `Intents` und speichert diese für die weitere Verwendung im `Intent`-Objekt ab. An dieser Stelle wird überprüft, ob der Benutzer oder die Benutzerin über die nötigen Rechte verfügt, um die Ziel-Komponente auszuführen und wenn ja, ob der Applikations-Prozess bereits läuft. Es werden drei verschiedene Phasen während des Starts eines neuen Applikations-Prozesses unterschieden:

1. Prozess erzeugen
2. Applikation binden
3. Aktivität, Dienste oder Broadcast Receiver starten

1. Prozess erzeugen

Der `Activity Manager Service` erzeugt einen neuen Prozess, indem er die Methode `startProcessLocked()` aufruft, welche über eine Socket-Verbindung Argumente an den

Zygote Prozess schickt. Dieser forkt sich anschließend selbst, startet den `ActivityThread` und gibt den Process Identifier (PID) des neu erstellten Prozesses zurück.

2. Applikation binden

Im nächsten Schritt wird die Applikation an den Prozess gebunden. Dies geschieht, indem eine `BIND_APPLICATION` Nachricht an den `ActivityThread`, beziehungsweise an die darin ausgeführte `Looper`-Instanz, geschickt wird. Diese gibt die Nachricht an `handleMessage()` und in weiterer Folge an `handleBindApplication()` weiter, woraufhin ein Aufruf von `makeApplication()` folgt. An dieser Stelle werden schließlich die applikationsspezifischen Klassen geladen.

3. Aktivität starten

Nach dem Binden der Applikation an den Prozess befinden sich die Klassen der Anwendung im privaten Speicher des Prozesses. Das Starten einer Aktivität unterscheidet sich von hier an nicht, egal ob es sich um einen neu erstellten oder einen bereits laufenden Prozess handelt. Dazu wird vom `Activity Manager Service` die `LAUNCH_ACTIVITY` Nachricht an den `ActivityThread` gesendet. Diese Nachricht wird durch den `Looper` von der `handleLaunchActivity()` Methode entgegengenommen, welche die entsprechende Aktivität schließlich startet.

Lebenszyklus einer Aktivität

Aktivitäten werden unter Android durch einen `Activity Stack` verwaltet. Die Oberste ist immer die gerade laufende Aktivität (Google 2014a). Wie in Abbildung 4 zu sehen ist, kann eine Aktivität dabei immer nur einen von vier Zuständen aufweisen:

- **laufend:** Wenn sich eine Aktivität am Bildschirm im Vordergrund befindet, ist sie aktiv oder läuft. Dieser Aktivität wird vom System die höchste Priorität zugewiesen und nur in Notfällen, etwa wenn sie mehr Speicher verbraucht als vom System zur Verfügung gestellt werden kann, beendet.
- **pausiert:** Wenn eine Aktivität den Fokus verloren hat, aber trotzdem noch sichtbar ist, beispielsweise wenn sich eine transparente Aktivität oder ein Dialogfeld darüber befindet, wird sie pausiert. Anders als pausiert vermuten lässt, bleibt die Aktivität allerdings vollständig aktiv, sie ist ja auch noch teilweise sichtbar. Eine pausierte Aktivität wird deshalb vom System nur bei extremer Ressourcenknappheit beendet.
- **gestoppt:** Wenn eine Aktivität vollkommen von einer anderen überdeckt wird, geht sie in den Zustand „gestoppt“ über. Sie speichert den aktuellen Zustand und kann somit vom System sicher beendet werden, wenn an anderer Stelle Ressourcen benötigt werden.

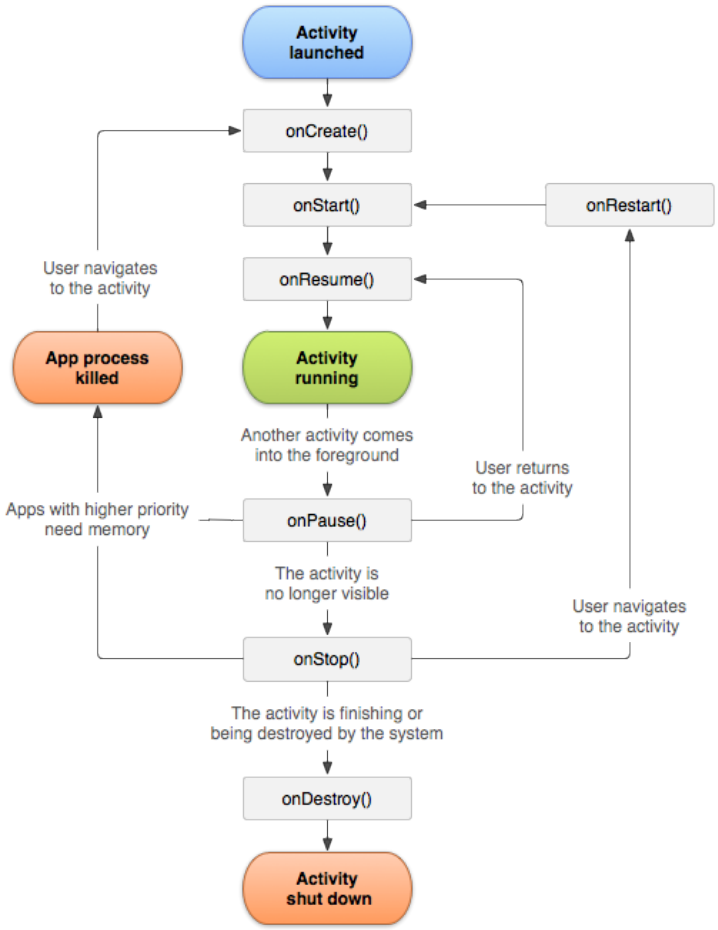


Abbildung 4: Lebenszyklus einer Aktivität (Google 2014a)

- **beendet:** Wenn sich eine Aktivität im Zustand „pausiert“ oder „gestoppt“ befindet, kann sie vom System beendet werden. Wenn die beendete Aktivität allerdings am `Activity Stack` wieder an vorderste Stelle rückt, muss sie neu gestartet und ihr vorheriger Zustand wiederhergestellt werden.

Abbildung 4 zeigt ein Diagramm mit den wichtigsten Zuständen einer Aktivität und ihren möglichen Transitionen.

2.4 Android Sicherheit

Bei einer offenen Plattform wie Android stellt die Sicherheit eine besondere Herausforderung dar. Alle Änderungen am Android Quellcode unterlaufen deshalb dem „Android Security Program“ (Google 2014e). Dieses beinhaltet unter anderem folgende Schritte:

- **Penetrationstests und Code-Reviews:** Während der Entwicklung von Android Komponenten, werden diese einer Vielzahl von Sicherheitsüberprüfungen unterzo-

gen. Diese werden vom „Android Security Team“ und dem „Information Security Engineering Team“ von Google, sowie von unabhängigen Sicherheits-Beratern durchgeführt. Ziel ist es, mögliche Schwachstellen bereits vor der Veröffentlichung des Quellcodes zu beheben.

- **Open Source und Community-Review:** Das AOSP ermöglicht eine umfassende Sicherheitsüberprüfung durch die Community. Android verwendet auch andere quelloffene Technologien wie den Linux Kernel, der weitgehenden Sicherheitsüberprüfungen unterzogen wurde.
- **Laufende Überwachung:** Trotz dieser Vorkehrungen kann es nach dem Ausliefern der Software zu Problemen kommen. Deshalb überwacht das „Android Security Team“ Diskussionen zu Android-spezifischen oder allgemeinen Sicherheitsrisiken, um auf potentielle Gefahren und Schwachstellen schnellstmöglich zu reagieren.

Das Android Betriebssystem versucht, personenbezogene Daten und System-Ressourcen bestmöglich zu schützen, sowie Applikationen voneinander zu isolieren. Um diese Ziele zu erreichen, setzt Android auf fünf Sicherheits-Bestandteile:

1. Robuste Sicherheit auf Betriebssystem-Ebene durch den Linux Kernel
2. Obligatorische Applikations-Sandbox für alle Anwendungen
3. Sichere Kommunikation zwischen Prozessen
4. Signierung und Verifizierung von Applikationen
5. Anwendungsdefinierte, von BenutzerInnen gewährte Berechtigungen

Auf der Ebene des Betriebssystems baut die Android Plattform auf der Sicherheit des Linux Kernels, sowie einer Android-spezifischen Schnittstelle zur prozessübergreifenden Kommunikation namens Binder auf. Diese IPC Schnittstelle ermöglicht es Applikationen, die in verschiedenen Prozessen ausgeführt werden, zu kommunizieren.

Auch nativer Code wird von der Applikations-Sandbox berücksichtigt. Wenn es sich dabei um schädlichen Code handelt, sollte dieser nicht auf andere Anwendungen oder das System übergreifen können (Google 2014e).

In den folgenden Unterkapiteln wird auf diese Bestandteile, sowie auf bestehende Arbeiten, die sich näher mit der Sicherheit auseinandersetzen, eingegangen.

2.4.1 System- und Kernel-Sicherheit

Die Basis der Android Plattform stellt der Linux Kernel dar. Dieser existiert seit Anfang der 1990er Jahre und wird in Millionen von sicherheitskritischen Umgebungen eingesetzt. In seiner Geschichte wurde er von tausenden EntwicklerInnen untersucht, attackiert und

repariert. Dadurch entwickelte er sich zu einem stabilen und sicheren Kernel, dem viele Firmen und Sicherheits-Experten vertrauen (Google 2014e).

Der Linux Kernel stellt der Android Plattform einige wichtige sicherheitsrelevante Funktionen zur Verfügung, unter anderem:

- Ein Berechtigungsmodell auf BenutzerInnen- und Gruppen-Ebene
- Isolation von Prozessen
- Erweiterbare Mechanismen zur prozessübergreifenden Kommunikation
- Die Möglichkeit, nicht benötigte und potentiell unsichere Komponenten aus dem Kernel zu entfernen

Als ein MehrbenutzerInnen-Betriebssystem ist es eines der fundamentalen Ziele des Linux Kernels, Ressourcen von BenutzerInnen voneinander zu isolieren. Er stellt sicher, dass:

- BenutzerIn A nicht auf Dateien von BenutzerIn B zugreifen kann
- BenutzerIn A nicht den Hauptspeicher von BenutzerIn B verbraucht
- BenutzerIn A nicht die CPU-Ressourcen von BentzerIn B erschöpft
- BenutzerIn A nicht die Geräte (zum Beispiel GPS) von BenutzerIn B blockiert

Vor Android 4.3 beruhte diese Isolation auf Discretionary Access Control (DAC). Dabei handelt es sich um ein Sicherheitskonzept, bei dem die Entscheidung, ob auf eine Ressource zugegriffen werden darf oder nicht, auf BenutzerInnen- und Gruppen-Basis getroffen wird.

DAC weist einige gravierende Nachteile auf. Darunter die Möglichkeit, dass fehlerhafte oder bösartige Applikationen Daten durchsickern lassen können, die grobe Granularität der DAC-Berechtigungen und das Unvermögen, Systemprogramme zu isolieren, die als Root ausgeführt werden (Smalley und Craig 2013).

SELinux

SELinux wurde ursprünglich von der National Security Agency als Linux Security Modul entwickelt. Es beinhaltet unter anderem einen Mandatory Access Control (MAC) Mechanismus, der die Schwächen von DAC überwinden soll. Im Gegensatz zu DAC erzwingt SELinux eine systemweite Sicherheits-Policy, die alle Prozesse, Objekte und Operationen einschließt. Durch den MAC Mechanismus ist es SELinux möglich, schädliche und fehlerhafte Anwendungen, sogar diejenigen, die als Root oder Superuser ausgeführt werden, zu isolieren (Smalley und Craig 2013).

Ohne SELinux wäre Android ein leichtes Ziel für Angriffe auf System-Services, die als Root ausgeführt werden. Würde ein Angreifer dort eine Schwachstelle finden, könnte er

die volle Kontrolle über das Endgerät erlangen. Die Community hat im Laufe der Zeit bereits einige kritische Sicherheitsschwachstellen in Android gefunden. Diese Schwachstellen reichten von Buffer-Overflows in frühen Versionen des Applikationsframeworks bis hin zur Interpretation von allen lokalen Tasteneingaben auf G1 Smartphones als Root-Shell Kommandos (Shabtai, Fledel und Elovici 2010).

Beginnend mit Android 4.3 wurde SELinux in den Android Kernel aufgenommen, um dessen Sicherheit zu erhöhen. Zum Beispiel muss eine Anwendung typischerweise über Root-Rechte verfügen, um auf die System-Partition zu schreiben. Wenn in einer traditionellen DAC basierten Linux-Umgebung, wie es vor Android 4.3 der Fall war, der Root-Benutzer kompromittiert wurde, konnte eine Applikation Zugriff auf alle Systemdateien erlangen. SELinux kann in so einem Fall den Zugriff mit Hilfe einer Policy trotzdem verhindern. Solch eine Policy könnte beispielsweise besagen, dass Anwendungen von Drittanbietern, auch wenn sie als Root ausgeführt werden, keinen Zugriff auf Systemdateien erhalten (Google 2014e).

2.4.2 Applikations-Sandbox

Einer der Grundpfeiler für die Sicherheit unter Android ist die sogenannte Applikations-Sandbox. Darunter ist die Isolation von Ressourcen auf BenutzerInnen und Gruppenebene durch den Linux Kernel gemeint.

Das Android System weist jeder Applikation während der Installation einen eindeutigen User Identifier (UID) zu. Jede Applikation wird in weiterer Folge immer in einem eigenen Prozess, dem diese UID zugewiesen ist, ausgeführt. Dieser Ansatz unterscheidet sich von anderen Betriebssystemen (einschließlich der traditionellen Linux-Distributionen) dahingehend, dass normalerweise mehrere Anwendungen einem oder einer BenutzerIn zugewiesen sind. Dadurch entsteht eine Applikations-Sandbox auf Level des Kernels. Standardmäßig können Applikationen nicht untereinander kommunizieren und haben nur begrenzten Zugriff auf das Betriebssystem. Wenn Anwendung A etwas Bösartiges versucht, wie etwa auf die Daten von Anwendung B zuzugreifen oder einen Anruf zu tätigen, wofür wiederum eine separate Anwendung C nötig ist, wird das vom Kernel verhindert, wenn Anwendung A nicht über die nötigen Rechte verfügt. Da sich diese Anwendungs-Sandbox im Kernel befindet, greift sie auch bei nativen Code und System-Anwendungen. Jegliche Software oberhalb des Kernels, wie in Abbildung 1 zu sehen ist, einschließlich System-Bibliotheken, Applikationsframework und die Android Runtime samt aller Applikationen, wird in dieser Sandbox ausgeführt. Auf einigen Betriebssystemen können Speicherzugriffsverletzungen die Sicherheit des gesamten Systems beeinträchtigen. Das ist unter Android nicht der Fall, da sich alle Applikationen und deren Ressourcen in einer Sandbox auf Betriebssystem-Ebene befinden. Eine Speicherzugriffsverletzung erlaubt nur das Ausführen von Code im Kontext der jeweiligen Applikation, unter Berücksichtigung der ihr gewährten Berechtigungen. Wie jedes Sicherheits-Feature ist auch die Android Applikations-Sandbox nicht unüberwindbar. Um jedoch aus der Sandbox eines standardmäßig konfigurierten Android Systems auszubrechen, muss der Linux Kernel kompromittiert werden (Google 2014e).

2.4.3 Prozessübergreifende Kommunikation

IPC ist auf Android über die herkömmlichen UNIX Mechanismen, wie das Dateisystem, lokale Sockets oder Shared Memory möglich. Diese Mechanismen sollten allerdings nur in Ausnahmefällen direkt von EntwicklerInnen verwendet werden, weil dadurch Android Sicherheits-Mechanismen umgangen werden. Android stellt deshalb für die Kommunikation zwischen verschiedenen Komponenten eigene Mechanismen zur Verfügung:

- **Binder:** Dabei handelt es sich um einen leichtgewichtigen, hoch performanten RPC Mechanismus.
- **Dienste:** Dienste können ein Interface bereitstellen, welches direkt durch Binder angesprochen werden kann.
- **Intents:** Ein Intent ist eine Nachricht, die über die Binder IPC Schnittstelle gesendet wird, siehe Kapitel 2.2.6.
- **Content Provider:** Content Provider sind für die Verwaltung der Daten einer Applikation verantwortlich und können diese auch anderen Prozessen oder Applikationen zur Verfügung stellen, siehe Kapitel 2.3.1.

Wie in der obigen Auflistung zu sehen ist, spielt Binder für die IPC auf Android eine zentrale Rolle. Binder wurde als Kernel Modul implementiert und kann durch `/dev/binder` angesprochen werden. Dieser Endpunkt muss von allen Prozessen beschreibbar sein, wodurch es dem Linux Kernel nicht möglich ist, die Binder IPC durch DAC zu überwachen. Deshalb wird jegliche Kommunikation über Binder durch das Android Framework mittels eines MAC Referenz Monitors überwacht. Jede Applikation oder Applikations-Komponente verfügt dafür über bestimmte Berechtigungen, die in ihrer Manifest-Datei festgelegt werden. Wenn eine Komponente eine IPC initiiert, wird überprüft, ob die entsprechende Berechtigung vorhanden ist. Wenn ja, wird die IPC fortgesetzt, andernfalls abgebrochen, selbst wenn sich die Komponenten in derselben Applikation befinden (Enck, Ongtang und McDaniel 2009).

In (Chin u. a. 2011) beschreiben die Autoren, welches Gefahrenpotential in den IPC Mechanismen von Android steckt, wenn EntwicklerInnen es verabsäumen, Sicherheitsvorkehrungen zu treffen. Hinausgehende Kommunikation kann zum Beispiel zu Broadcast- und Datendiebstahl, der Modifikation von IPC Ergebnissen oder Aktivitäts- und Dienst-Hijacking führen. Hereinkommende Kommunikation kann eine Applikation der Gefahr von schädlichen Aktivitäten, dem ungewollten Start von Diensten oder einer Broadcast Injektion aussetzen.

Wie in (Davi u. a. 2011) und (Felt, Wang u. a. 2011) beschrieben wird, kann es trotz der Applikations-Sandbox zu einer *Confused Deputy Attack* beziehungsweise *Permission Re-Delegation* kommen. Dies kann der Fall sein, wenn einzelne Komponenten einer Applikation nicht ordnungsgemäß vor der Verwendung von nicht autorisierten Anwendungen

geschützt wurden. Die Android Sicherheits-Architektur verhindert nicht, dass Komponenten, welche bestimmte Berechtigungen benötigen, von externen Applikationen aufgerufen werden, die nicht mindestens über dieselben Berechtigungen verfügen.

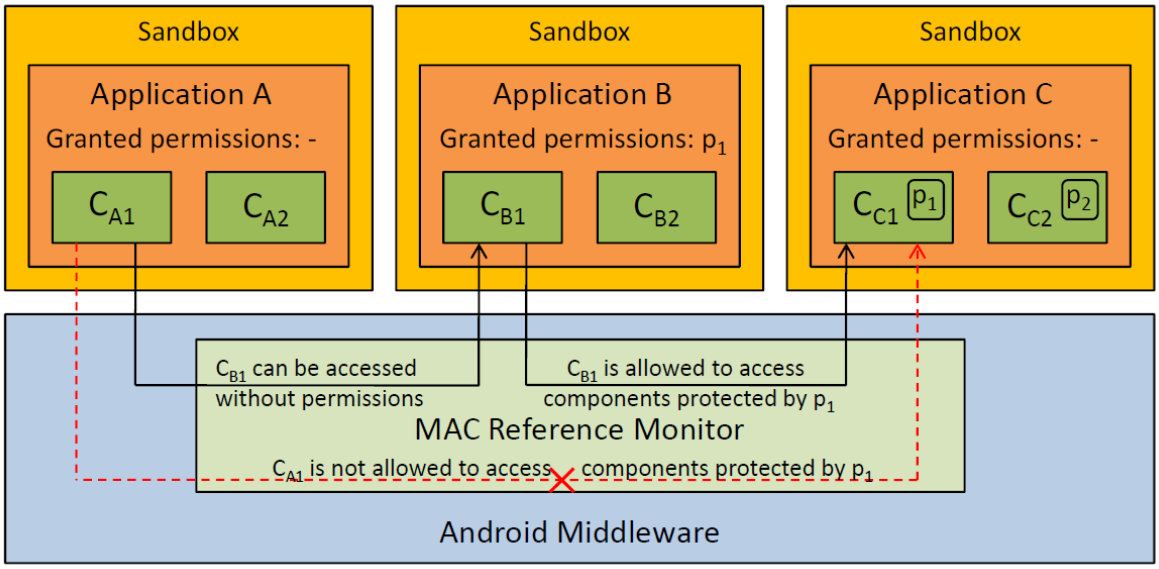


Abbildung 5: Permission Re-Delegation Attack (Davi u. a. 2011)

In Abbildung 5 wird eine mögliche *Permission Re-Delegation Attack* beschrieben. Hier kann Applikation A ohne besondere Berechtigungen eine Komponente von Applikation B nutzen. Applikation B wiederum verfügt über die Berechtigung, auf eine geschützte Komponente von Applikation C zuzugreifen. In weiterer Folge kann Applikation A diesen Umstand ausnutzen, um auf die geschützte Komponente von Applikation C zuzugreifen.

In (Bugiel u. a. 2011) wird ein Framework namens XManDroid (eXtended Monitoring on Android) vorgestellt, das in der Lage sein soll, *Privilege Escalation Attacks*, worunter auch *Permission Re-Delegation* oder *Confused Deputy Attacks* fallen, zu verhindern. Das Framework erweitert dafür die Überwachungs-Mechanismen von Android dahingehend, dass auch der Gebrauch von transitiven Berechtigungen mittels MAC erkannt wird. Diese Arbeit wurde später in (Bugiel u. a. 2012) fortgesetzt.

Da es sich bei *Privilege Escalation Attacks* allerdings um ein Schlupfloch in der Android Sicherheits-Architektur handelt, beziehungsweise um einen Mangel, der durch die fehlerhafte Implementierung von Applikations-Komponenten durch EntwicklerInnen hervorgerufen wird, kann auch MAC nicht immer davor schützen.

2.4.4 Applikations-Signierung

Jede Applikation, die auf Android installiert werden soll, muss durch deren EntwicklerInnen signiert werden. Applikationen, die nicht signiert wurden, werden standardmäßig bei

der Installation entweder von Google Play oder dem Package Manager zurückgewiesen. Das gewährleistet einerseits, dass Applikationen unverändert auf den Endgeräten ankommen, andererseits können so EntwicklerInnen von Google für schädliches Verhalten ihrer Anwendungen zur Verantwortung gezogen werden.

Die Signierung einer Applikation ist der erste Schritt, um eine Anwendung in der Android Sandbox auszuführen. Das Zertifikat entscheidet darüber, welche UID einer Applikation zugewiesen wird. Während der Installation überprüft der Package Manager, ob die zu installierende APK mit dem in ihr enthaltenen Zertifikat signiert wurde. Wenn ein Zertifikat für die Signierung mehrerer Applikationen verwendet wurde, besteht die Möglichkeit, dass sich diese Applikationen eine UID und somit auch eine Sandbox teilen. Da normalerweise jede Android Applikation in ihrer eigenen Sandbox ausgeführt wird und es sich dabei um einen der grundlegenden Sicherheitsmechanismen unter Android handelt, muss jede beteiligte Applikation diese Ausnahme im Manifest deklarieren (Google 2014e).

Nur weil eine Applikation signiert wurde, bedeutet dies allerdings nicht, dass sie nicht verändert wurde. Wie (Zhou, Zhou u. a. 2012) beschreiben, sind 5-13% aller Applikationen in alternativen Marktplätzen „repackaged applications“. Dabei handelt es sich meist um legitime, oft kostenpflichtige Applikationen aus dem offiziellen *Google Play Store*, die modifiziert wurden. Diese Modifikationen reichen vom Hinzufügen oder Austauschen von Werbebannern bis hin zum Hinzufügen von Schadcode. Die Applikationen werden nach der Modifikation zu neuen APK Dateien gepackt und neu signiert.

2.4.5 Applikations-Verifizierung

Ein Bestandteil, der nicht direkt mit dem Android Betriebssystem zusammenhängt, aber dennoch für den Schutz von EndanwenderInnen vor Schadsoftware eine wichtige Rolle spielt, ist eine Software namens Bouncer. Dabei handelt es sich um einen in den *Google Play Store* integrierten Sicherheitsmechanismus, der verhindern soll, dass bösartige Applikationen in den offiziellen Marktplatz gelangen. Bouncer sucht dafür automatisch nach schädlichen Anwendungen und erspart EntwicklerInnen damit ein aufwendiges Review-Verfahren beim Veröffentlichen ihrer Applikationen.

Der Dienst führt eine Reihe von Analysen auf neuen Applikationen, Applikationen, die sich bereits im Marktplatz befinden und auf EntwicklerInnen-Konten durch. Sobald eine Applikation in den Marktplatz hochgeladen wurde, startet der Dienst sofort mit der Suche nach bekannter Schadsoftware, Spyware und Trojanern. Anschließend wird eine dynamische Analyse der Applikation durchgeführt, und ihr Verhalten mit zuvor überprüften Anwendungen verglichen, um mögliche schädliche Verhaltensmuster zu identifizieren (Lockheimer 2012).

In (Oberheide und Miller 2012) und (Paleari u. a. 2009) werden Verfahren beschrieben, wie Bouncer umgangen werden kann. Wird eine Applikation dynamisch analysiert, kann beispielsweise festgestellt werden, ob sie auf einem echten Endgerät oder in einem Emulator ausgeführt wird. Wird ein Emulator erkannt, wird kein schädlicher Code ausgeführt und dieser auch nicht erkannt.

Seit Android 4.2 verfügt Android über ein Feature namens Applikations-Verifizierung. Dieses wurde in weiterer Folge für alle Android Versionen beginnend mit 2.3 nachgereicht. Die Applikations-Verifizierung wird besonders empfohlen, wenn Anwendungen von unbekannten Quellen, beispielsweise alternativen Marktplätzen, installiert werden. Ist dieses Feature aktiviert, wird jede Applikation vor ihrer Installation auf mögliche Schadsoftware überprüft (Google 2014e).

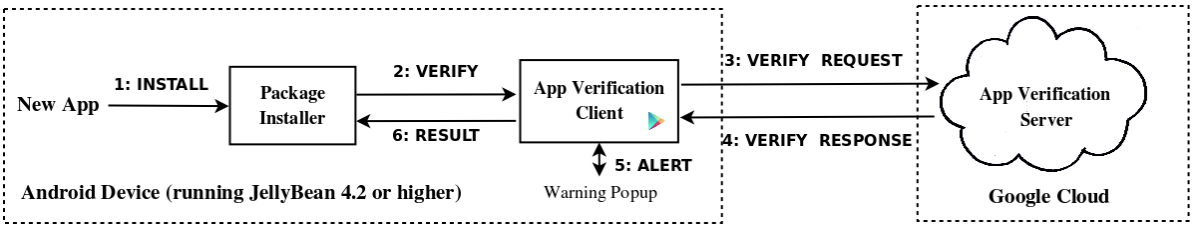


Abbildung 6: Applikations-Verifizierungs-Dienst von Google (Jiang 2012)

In Abbildung 6 wird die Funktionsweise des Applikations-Verifizierungs-Dienstes beschrieben. Wenn eine Applikation installiert wird (Schritt 1), werden Informationen über diese (zum Beispiel Name, Version und Größe), sowie über das Endgerät, gesammelt (Schritt 2) und an die Google Cloud gesendet (Schritt 3). Die Google Cloud antwortet anschließend mit dem Ergebnis der Analyse (Schritt 4). Wenn die Applikation als potentiell gefährlich oder gefährlich eingestuft wurde, wird eine Warnmeldung angezeigt (Schritt 5). Potentiell gefährliche Applikationen können von dem oder der BenutzerIn auf eigene Gefahr installiert werden. Die Installation von gefährlichen Applikationen wird blockiert (Schritt 6).

Aus einer Evaluierung (Jiang 2012) des Applikations-Verifizierungs-Dienstes geht hervor, dass dieser, kurz nach dessen Einführung, weniger als 25% der bekannten Schadsoftware Samples erkannte. Zum selben Zeitpunkt erzielten 9 von 10 getesteten Antivirus-Lösungen auf demselben Datensatz bereits eine Erkennungsrate von 75-100%.

Seit einem Update des *Google Play Store* im Februar 2014 läuft die Applikations-Verifizierung auf allen Android Versionen, beginnend mit Android 2.3, periodisch im Hintergrund und nicht mehr nur vor der Installation, was zusätzlichen Schutz gewährleisten soll (Whitwam 2014).

2.4.6 Berechtigungs-System

Das Berechtigungs-System stellt einen der wichtigsten Sicherheits-Mechanismen auf Android dar. Es steuert den Zugang zu datenschutz- und sicherheitsrelevanten Teilen der Android API. Jede Anwendung enthält in ihrer Manifest-Datei (siehe Kapitel 2.3.3) Informationen dazu, welche Berechtigungen sie benötigt, um ordnungsgemäß zu funktionieren (Google 2014e).

Solche sensible APIs, für die spezielle Berechtigungen benötigt werden, beinhalten unter anderem:

- Zugriff auf die Kamera zum Aufnehmen von Bildern und Videos

- Zugriff auf das Mikrofon
- Zugriff auf Positionsdaten (GPS)
- Zugriff auf Funktionen zum Telefonieren
- Zugriff auf das Senden und Empfangen von SMS/MMS
- Zugriff auf Netzwerk beziehungsweise Datenverbindungen (Internet)

Bei der obigen Auflistung handelt es sich um Berechtigungen, die unter die Kategorie „Gefährlich“ fallen. Es gibt noch zwei weitere Kategorien, „Normal“ und „Signatur/System“. Außerdem sind EntwicklerInnen von Android Applikationen in der Lage, eigene Berechtigungen zu definieren, um so den Zugriff auf ihre Anwendungen zu steuern (Felt, Chin u. a. 2011).

Normal

Normale Berechtigungen schützen BenutzerInnen vor API-Aufrufen, die zwar stören, allerdings keinen Schaden verursachen können. Ein Beispiel hierfür ist die Berechtigung `SET_WALLPAPER`, welche eine Applikation berechtigt, das Hintergrundbild eines Benutzers oder einer Benutzerin zu verändern.

Gefährlich

Gefährliche Berechtigungen schränken die Benutzung von potentiell gefährlichen APIs ein. Diese beinhalten beispielsweise den Zugriff auf personenbezogene Daten oder Funktionalität, die den Benutzer oder die Benutzerin Geld kosten kann. Dabei kann es sich etwa um das automatische Senden von SMS-Nachrichten oder den Zugriff auf das Adressbuch handeln.

Signatur/System

System- und Signatur-Berechtigungen steuern den Zugriff auf besonders gefährliche Teile der API, wie beispielsweise den Package Manager, mit dem Applikationen installiert oder gelöscht werden können. Diese Berechtigungen können deshalb auch nicht einfach, wie andere Berechtigungen, in der Manifest-Datei einer Applikation deklariert werden. Signatur-Berechtigungen werden nur Applikationen erteilt, die mit dem Zertifikat des Gerätehersteller signiert wurden. Um System-Berechtigungen zu erlangen, muss eine Applikation entweder vom Gerätehersteller signiert oder in ein spezielles System-Verzeichnis installiert worden sein. Dadurch wird die Verwendung von Signatur- oder System-Berechtigungen im Grunde auf vorinstallierte Applikationen limitiert, solange es sich nicht um ein „gerootetes“ Endgerät handelt. Wurde ein Endgerät „gerootet“, können auch Applikationen von Drittanbietern mit Root-Rechten ausgeführt werden, womit vorhandene Sicherheits-Mechanismen umgangen werden.

Applikations-Berechtigungen

Auch EntwicklerInnen können eigene Berechtigungen für ihre Applikationen definieren. Sie sind somit in der Lage, den Zugriff auf deren Funktionalität oder Daten zu schützen. Meist werden solche Berechtigungen auf Signatur-Basis definiert, um nur Applikationen derselben EntwicklerInnen Zugriff darauf zu erlauben.

BenutzerInnen werden dazu aufgefordert, diese Berechtigungen vor der Installation einer jeden Anwendung, beziehungsweise beim Updaten, wenn sich an den Berechtigungen etwas geändert hat, zu bestätigen. Wie Abbildung 7 zeigt, liegt es in der Verantwortung der BenutzerInnen, ob sie einer Anwendung vertrauen oder die Installation anderenfalls abbrechen (Google 2014e).

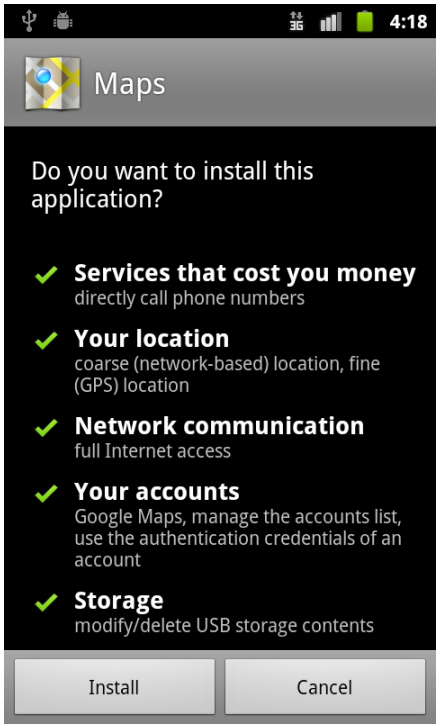


Abbildung 7: Berechtigungs Dialog bei der Installation einer Applikation (Google 2014e)

Es ist jedoch fraglich, wie aufmerksam BenutzerInnen lesen, welche Berechtigungen von einer Applikation tatsächlich angefordert werden. Einer Studie zufolge (Felt u. a. 2012) beachteten nur 17% der Teilnehmer während der Installation von Applikationen die angeforderten Berechtigungen und nur 3% konnten drei Fragen zu diesen richtig beantworten. Auch wenn sich laut der Autoren nur knapp ein Fünftel der Probanden mit den Applikations-Berechtigungen auseinandergesetzt hat, könnten diese ausreichen, um andere BenutzerInnen zu schützen, wenn ihre Meinung über bestimmte Applikations-Berechtigungen erfolgreich über Applikations-Reviews kommuniziert werden kann.

Ein großes Problem ist laut (Felt, Chin u. a. 2011) allerdings die Tatsache, dass gerade unerfahrene EntwicklerInnen routinemäßig mehr Berechtigungen anfordern als sie tat-

sächlich benötigen. In einer statischen Analyse von 900 Android Applikationen fanden die Autoren, dass 323 (35,8%) über-privilegiert waren.

Da es sich hierbei, neben der Applikations-Sandbox, um einen der Grundpfeiler der Android Sicherheit handelt, beschäftigen sich bereits einige Arbeiten mit diesem Thema. Es wurden verschiedene Vorschläge erarbeitet, wie das Berechtigungs-System und der Zugriff auf APIs und Ressourcen auf Android verbessert werden könnten. Darunter finden sich unter anderem:

- **Apex:** Bei Apex handelt es sich um eine Erweiterung des Android Berechtigungs-Systems. Es erlaubt es BenutzerInnen, genau festzulegen, auf welche sensiblen Ressourcen eine Applikation Zugriff haben soll. Die Autoren bemängeln, dass BenutzerInnen unter Android bei der Installation einer Applikation zwar alle benötigten Berechtigungen aufgelistet bekommen, es aber nicht möglich ist, nur bestimmte Berechtigungen zu gewähren. Außerdem gibt es keine Möglichkeit, die Verwendung von bestimmten Ressourcen mittels Umgebungsvariablen, wie beispielsweise der momentanen GPS Position des Endgerätes, einzuschränken. Um diese Funktionalität zu bieten, wurden das bestehende Berechtigungs-System und die Anwendung zum Installieren von neuen Applikationen dementsprechend erweitert (Nauman, Khan und Zhang 2010).

- **Saint:** Das Saint Framework bietet eine erhöhte Flexibilität und Sicherheit für die Einschränkung von Applikations-Interaktionen. Dafür schränkt Saint nicht nur die Vergabe von bestimmten Berechtigungen während der Installation einer Applikation ein, sondern ermöglicht auch die Aufstellung von Regeln für die Kommunikation zwischen Applikationen. Dadurch können beteiligte Applikationen sicherstellen, dass ihre Kommunikation unter sicheren Bedingungen stattfindet. Beispielsweise könnte eine Einkaufs-Applikation über eine Liste von vertrauenswürdigen Bezahl-Applikationen verfügen. Ob eine Applikation vertrauenswürdig ist, könnte in weiterer Folge von deren Version abhängig gemacht werden, um beispielsweise Versionen der Anwendung mit bekannten Sicherheitslücken auszuschließen (Ongtang u. a. 2012).

- **TISSA:** In (Zhou u. a. 2011) beschreiben die Autoren einen „Privacy-Modus“ für Android. Dieser Privacy-Modus kann dafür genutzt werden, einer Anwendung den Zugriff auf private Informationen zu verweigern. Wenn ein Benutzer oder eine Benutzerin eine nicht vertrauenswürdige Applikation installiert, kann er oder sie genau festlegen, welche privaten Informationen (beispielsweise Kontakte, Position oder Bilder) der Applikation zur Verfügung gestellt werden. Außerdem sind BenutzerInnen in der Lage, diese Einschränkungen zur Laufzeit anzupassen.

- **MockDroid:** Auch bei MockDroid (Beresford u. a. 2011) handelt es sich um eine modifizierte Version des Android Systems, die es erlaubt, den Zugriff auf bestimmte Berechtigungen zur Laufzeit zu unterbinden. Bei der Installation einer Anwendung müssen dafür nicht alle angeforderten Berechtigungen erlaubt werden. Immer, wenn

eine Applikation eine Ressource anfordert, für die keine Berechtigung erteilt wurde, wird diese vom System als leer oder nicht verfügbar zurückgegeben. Der Benutzer oder die Benutzerin wird in diesem Fall informiert und kann bei Bedarf die fehlende Berechtigung erteilen. Auf diese Art können BenutzerInnen während der Nutzung einer Applikation abwägen, ob die Funktionalität die Herausgabe von sensiblen Informationen rechtfertigt.

3 Methoden

In diesem Kapitel werden verschiedene Arten von Android Schadsoftware und deren Erkennung beschrieben. Dazu werden verschiedene Herangehensweisen zum Extrahieren von Features aus Android Anwendungen besprochen und auf die Analyse dieser mittels maschinellen Lernens eingegangen. Besonderes Augenmerk liegt auf künstlichen neuronalen Netzen, Selbst-organisierenden Karten und Counterpropagation Artificial Neural Networks.

3.1 Android Schadsoftware

Laut (Alcatel-Lucent 2014) steigt die Anzahl von mit Schadsoftware infizierten mobilen Endgeräten seit 2012 jährlich um etwa 20%. Seit Juni 2012 wurden fast 900.000 Malware-Samples identifiziert.

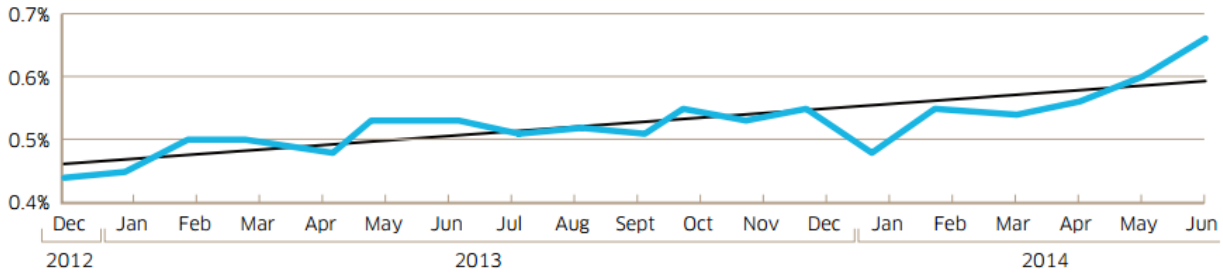


Abbildung 8: Infektionsrate mobiler Endgeräte (Alcatel-Lucent 2014)

Abbildung 8 zeigt den monatlichen Prozentsatz von infizierten mobilen Endgeräten seit Dezember 2012. Zu sehen ist ein jährlicher Anstieg von etwa 20%. In der ersten Hälfte von 2014 hat sich dieser Anstieg allerdings fast verdoppelt. Die Infektionsrate von 0,65% entspricht etwa 15 Millionen infizierten Endgeräten, bei geschätzten 2,3 Milliarden Smartphones weltweit. Da diese Schätzung von (Alcatel-Lucent 2014) keine Daten zu Regionen wie China und Russland enthält, wo Infektionsraten laut der AutorInnen tendenziell höher ausfallen, entsprechen diese Zahlen eher einer unteren Grenze.

3.1.1 Arten von Schadsoftware

BenutzerInnen von mobilen Endgeräten werden laut (Felt, Finifter u. a. 2011) von drei Arten mehr oder weniger schädlicher Software bedroht: *Malware*, *Grayware* und *Personal Spyware*. Diese drei Kategorien ergaben sich für die AutorInnen aufgrund der Art der Auslieferung von Schadsoftware, der Legalität der Software selbst und ob EndbenutzerInnen über die genaue Funktionalität dieser Software in Kenntnis gesetzt werden.

Malware

Malware verschafft sich Zugriff zu ansonsten geschützten Funktionen des Endgerätes, um beispielsweise Daten zu stehlen oder dem oder der BenutzerIn finanziellen Schaden zuzufü-

gen. Der oder die AngreiferIn bringt BenutzerInnen ohne deren Wissen oder Einwilligung dazu, schädliche Applikationen zu installieren, oder verschafft sich unbefugten Zugriff auf deren Endgeräte, indem er oder sie eine Sicherheitslücke ausnutzt. *Malware* inkludiert Trojaner, Würmer, Bot-Netze und Viren. Diese sind in den meisten Ländern verboten und deren Einsatz kann zu Gefängnisstrafen führen (Felt, Finifter u. a. 2011).

Personal Spyware

Personal Spyware sammelt über einen bestimmten Zeitraum personenbezogene Daten, wie beispielsweise den Aufenthaltsort (GPS Koordinaten) oder SMS-Nachrichten von BenutzerInnen. Um *Personal Spyware* einzusetzen, benötigten AngreiferInnen physischen Zugriff auf das zu überwachende Endgerät, um die Schadsoftware ohne das Wissen der Opfer zu installieren. *Personal Spyware* sendet die gesammelten Informationen direkt an den oder die AngreiferIn und nicht an die EntwicklerInnen der Software selbst. Zum Beispiel könnte ein Unternehmen *Personal Spyware* auf den Dienst-Telefonen der Mitarbeiter installieren. In einigen Ländern, wie beispielsweise den USA, ist es erlaubt *Personal Spyware* zu verkaufen, da KäuferInnen (AngreiferInnen) nicht selbst geschädigt werden und ihnen gegenüber der Zweck der Anwendung nicht verschleiert wird. Es ist allerdings illegal, *Personal Spyware* auf dem Smartphone einer anderen Person ohne deren Einwilligung zu installieren (Felt, Finifter u. a. 2011).

Grayware

Viele legitime Applikationen sammeln Informationen über deren BenutzerInnen, um Profile zu erstellen. Diese werden beispielsweise eingesetzt, um den BenutzerInnen personalisierte Werbung anzuzeigen. *Grayware* spioniert zwar BenutzerInnen aus, zielt allerdings nicht darauf ab, Schaden anzurichten. In vielen Fällen bietet diese sogar echte Funktionalität, wie zum Beispiel das Präsentieren oder Vorschlagen von auf BenutzerInnen zugeschnittene Inhalte. Unternehmen, die *Grayware* vertreiben, müssen darauf in ihrer Datenschutzerklärung hinweisen. *Grayware* befindet sich oftmals an der Grenze zur Illegalität, je nachdem wie detailliert die jeweilige Datenschutzerklärung ist beziehungsweise ob die darin enthaltenen Paragrafen rechtmäßig sind.

Wenn in dieser Arbeit von Schadsoftware gesprochen wird, liegt der Fokus vor allem auf verschiedenen Arten von *Malware*, also Trojanern, Würmern, Bot-Netzen und Viren.

Motivation hinter Schadsoftware

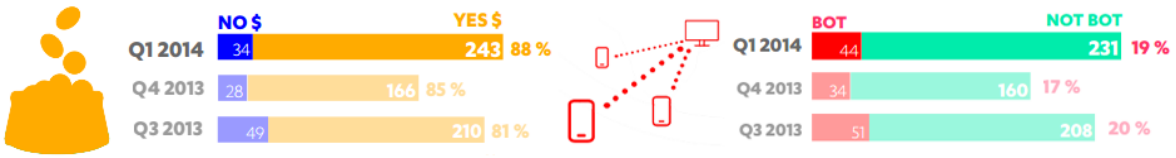


Abbildung 9: Die Motivation hinter Schadsoftware (F-Secure 2014)

Wie Abbildung 9 zeigt, ist die Motivation hinter mobiler Schadsoftware vor allem finanzieller Profit. (Juniper 2013) zufolge wird davon ausgegangen, dass jede veröffentlichte schädliche Applikation im Durchschnitt einen sofortigen Gewinn von 10 USD abwirft. 19 Prozent der von F-Secure im ersten Quartal 2014 analysierten Schadsoftware gliederte das Endgerät zusätzlich in ein Bot-Netz ein. Dadurch sind AngreiferInnen in der Lage, die Endgeräte über einen Command-and-Controll-Server zu steuern und in weiterer Folge für ihre Zwecke zu missbrauchen.

Laut (F-Secure 2014) stellen momentan Trojaner die größte Gefahr für mobile Endgeräte dar. Dabei wurden im ersten Quartal 2014 typischerweise eine oder mehrere der folgenden Aktivitäten ausgeführt:

- **SMS-Betrug:** SMS-Nachrichten werden an Mehrwertnummern gesendet oder der oder die BenutzerIn wird bei auf SMS-Nachrichten basierenden Abo-Diensten angemeldet.
- **Applikations-Download:** Unerwünschte Applikationen werden im Hintergrund, ohne das Wissen des oder der BenutzerIn, heruntergeladen und installiert.
- **Datendiebstahl:** Diebstahl von Kontakten, Fotos und anderen personenbezogenen Daten wie beispielsweise der GPS Position.
- **Link-Klicker:** Im Hintergrund werden automatisch Websites aufgerufen und/oder auf Werbebanner geklickt, um so Umsätze zu generieren.
- **Bank Betrug:** Online-Banking bezogene SMS-Nachrichten werden mitgelesen.
- **Fake Antivirus:** Eine Applikation gibt vor, eine mobile Antivirus Lösung zu sein, besitzt jedoch keine entsprechende Funktionalität.
- **Einheben von Gebühren:** Es werden für die Benutzung, das Aktualisieren oder die Installation einer legitimen (meist freien) Applikation Gebühren verrechnet.

3.1.2 Installations-Methoden

Um mobile Endgeräte zu infizieren, werden BenutzerInnen oftmals dahingehend getäuscht, dass sie schädliche Applikationen selbst herunterladen und installieren. (Jiang und Zhou 2013) teilten nach der Analyse von 1260 Schadsoftware Samples die Installations-Methoden in drei Haupt-Kategorien ein:

1. **Repackaging:** Dabei handelt es sich um eine der am weitesten verbreiteten Techniken, die EntwicklerInnen von Schadsoftware verwenden. Dazu wird schädlicher Code in legitimen Applikationen versteckt. Der oder die AngreiferIn muss dazu eine legitime Applikation herunterladen, diese de-kompilieren, den Schadcode hinzufügen und

die Applikation anschließend zu einer neuen APK kompilieren und diese neu signieren. Um nicht aufzufallen, wird der Schadcode oftmals in harmlos klingenden Klassen und Namensräumen versteckt, beispielsweise `com.sec.android.provider.drm`. Danach wird die modifizierte Applikation im offiziellen *Google Play Store* oder einem alternativen Marktplatz veröffentlicht. Laut (Jiang und Zhou 2013) handelte es sich bei 1083, der von ihnen analysierten 1260 schädlichen Applikationen (86%) um solche modifizierten Versionen von legitimen Anwendungen. Von den 49 Schadsoftware-Familien in ihrem Datensatz, nutzten 25 diese Technik, um Endgeräte zu infizieren.

2. **Update Attack:** Bei der ersten Technik wird einer Applikation der gesamte Schadcode eingepflanzt, der nötig ist, um einen Angriff zu starten. Dadurch besteht ein hohes Risiko, dass der schädliche Code durch statische Analyse erkannt wird. Eine *Update Attack* macht diese Erkennung schwieriger. In den meisten Fällen wird auch hier Schadcode in legitimen Applikationen versteckt, jedoch nicht der gesamte, sondern nur eine Update Komponente, die den wirklichen Schadcode zur Laufzeit nachlädt. Im Datensatz des Android Malware Genome Project (Jiang und Zhou 2013) befinden sich vier Schadsoftware-Familien, die diese Technik nutzen: *BaseBridge*, *DroidKungFuUpdate*, *AnserverBot* und *Plankton*. Die ersten beiden benötigen das Einverständnis des oder der BenutzerIn, um die Applikation zu *aktualisieren*. Im Fall von *AnserverBot* und *Plankton* werden BenutzerInnen komplett umgangen. Anstelle der gesamten Applikation werden nur Teile ausgetauscht, wodurch auf dem Android System keine Zustimmung des oder der BenutzerIn benötigt wird. *Plankton* lädt dafür eine Java Archive (JAR)-Datei herunter und führt diese aus, während *AnserverBot* einen verschlüsselten Blog Eintrag herunterlädt, der den schädlichen Code enthält.

3. **Drive-by Download:** Dabei handelt es sich um eine Technik, bei der BenutzerInnen durch aggressive Werbung in Applikationen oder durch QR-Codes dazu bewegt werden, Schadsoftware zu installieren. Beispiele hierfür sind *GGTracker*, *Jifake*, *Spitmo* oder *ZitMo*. Bei der *GGTracker* Schadsoftware gelangen BenutzerInnen nach dem Klick auf einen Werbebanner auf eine Website, die dem *Google Play Store* nachempfunden wurde und werden aufgefordert, eine Applikation zu installieren. Bei *Jifake* geschieht dasselbe durch das Scannen eines QR-Codes. Beide Schadsoftware-Familien versenden fortan kostenpflichtige SMS-Nachrichten. Bei *Spitmo* und *ZitMo* handelt es sich um Schadsoftware, die unter die Rubrik Bank-Betrug fallen. In beiden Fällen muss allerdings der PC oder Laptop von BenutzerInnen von Online-Banking Diensten befallen sein. BenutzerInnen werden dazu verleitet, auf ihrem Smartphone eine Software zu installieren, welche die Sicherheit ihres Online-Bankings vermeintlich erhöhen soll (Jiang und Zhou 2013).

3.1.3 Aktivierungs-Methoden

Schadsoftware wird auf Android meist durch systemweite Events, durch das Austauschen der Haupt-Aktivität oder durch Eingaben von BenutzerInnen aktiviert. In vielen Fällen

Kategorie	Events
Systemstart	BOOT COMPLETED
Telefon	PHONE STATE NEW OUTGOING CALL
Packet Manager	PACKAGE ADDED PACKAGE REMOVED PACKAGE CHANGED PACKAGE REPLACED PACKAGE RESTARTED PACKAGE INSTALL
SMS	SMS RECEIVED WAP PUSH RECEIVED
USB	UMS CONNECTED UMS DISCONNECTED
Batterie	ACTION POWER CONNECTED ACTION POWER DISCONNECTED BATTERY LOW BATTERY OKAY BATTERY CHANGED ACTION
Netzwerk	CONNECTIVITY CHANGE PICK WIFI WORK
Applikation	ACTION MAIN
System	USER PRESENT INPUT METHOD CHANGED SIG STR SIM FULL

Tabelle 4: Auszug von Android Broadcast Events (Zhou und Jiang 2012)

nutzt Schadsoftware unter Android Broadcast Receiver (siehe 2.3.1), um den schädlichen Code im richtigen Moment ausführen zu können. In Tabelle 4 wurden einige der laut (Zhou und Jiang 2012) am meisten dafür genutzten Events aufgelistet.

Unter allen vorhandenen System-Events, ist `BOOT_COMPLETED` das wohl Interessanteste für Schadsoftware. Das ist wenig überraschend, da der Zeitpunkt, an dem das Android System den Startvorgang abgeschlossen hat, ein idealer Zeitpunkt ist, um schädliche Hintergrunddienste zu aktivieren. Durch das Hören auf solche Events kann Schadsoftware das Android Broadcast-System ausnutzen, um ohne die Interaktion von BenutzerInnen den Schadcode zum idealen Zeitpunkt auszuführen. Im Datensatz des Android Malware Genome Project befinden sich 29 von 49 Schadsoftware-Familien (oder 83,3% aller Samples), die auf das `BOOT_COMPLETED` Event hören. An zweiter Stelle kommt das `SMS_RECEIVED` Event mit 21 von 49 Schadsoftware-Familien, die darauf hören. Dieses Event wird vom Android System verbreitet, wenn eine neue SMS-Nachricht empfangen wurde. Schadsoftware ist damit in der Lage SMS-Nachrichten abzufangen, bevor sie dem oder der BenutzerIn angezeigt

werden, die SMS-Nachrichten zu löschen oder auch automatisch auf diese zu antworten. So wird Schadsoftware oftmals dazu genutzt, BenutzerInnen ohne deren Wissen bei Mehrwert-SMS-Diensten anzumelden (Zhou und Jiang 2012).

Mit Android 4.2 wurde ein Feature eingeführt, das Trojanern, die im Hintergrund Mehrwert SMS-Nachrichten versenden, entgegenwirken soll. BenutzerInnen wird immer dann ein Bestätigungs-Dialog angezeigt, wenn eine Applikation versucht, eine SMS-Nachricht an einen Mehrwert-Dienst zu senden. BenutzerInnen können dadurch selbst entscheiden, ob sie dieses Verhalten zulassen oder blockieren wollen (Google 2014e).

Es wurde auch beobachtet, dass einige Schadsoftware-Familien, beispielsweise *DroidDream*, die Haupt-Aktivität einer Applikation austauschen. Diese wird immer ausgeführt, wenn BenutzerInnen eine Applikation über deren Icon am Startbildschirm starten. Dadurch wird es der Schadsoftware ermöglicht, ihren schädlichen Code, noch bevor die eigentliche Applikation gestartet wird, auszuführen. Schädliches Verhalten kann außerdem durch Eingaben von BenutzerInnen ausgelöst werden. Ein Beispiel hierfür ist eine Schadsoftware-Familie namens *Zsone*. Diese erweitert beispielsweise den `onClick()`-Callback von GUI Elementen dahingehend, dass bei jedem Klick eine SMS-Nachricht gesendet wird (Zhou und Jiang 2012).

3.1.4 Distribution

Android BenutzerInnen können eine Vielzahl von Applikationen von Drittanbietern installieren. Diese beziehen sie meistens entweder aus dem offiziellen *Google Play Store* oder von einem der alternativen Marktplätze. Um Applikationen aus einem alternativen Marktplatz zu installieren, müssen BenutzerInnen in den Android Einstellungen die Option „Unbekannte Herkunft“ aktivieren und mit den damit verbundenen Risiken einverstanden sein. Ist diese Option aktiviert, sind BenutzerInnen in der Lage, APKs direkt zu installieren oder einen Marktplatz eines Drittanbieters zu nutzen.

Im Gegensatz zum offiziellen *Google Play Store*, der versucht, seine BenutzerInnen durch eine Software namens *Bouncer* vor Schadsoftware zu schützen (siehe 2.4.5), verfügen alternative Marktplätze oftmals über keine solchen Schutzmaßnahmen.

Laut (Juniper 2013) gibt es mehr als 500 alternative Marktplätze für Android Applikationen, die von Schadsoftware befallen sind. Diese Marktplätze von Drittanbietern können oftmals nur schwer zur Verantwortung gezogen werden, wodurch sich Schadsoftware dort sehr lange halten und zum Download angeboten werden kann. Das *Mobile Thread Center* von Juniper Networks zeigt außerdem, dass sich drei von fünf dieser alternativen Marktplätze in China und Russland befinden.

In einer Studie, die 2012 von (Zhou, Wang u. a. 2012) durchgeführt wurde und den *Google Play Store*, sowie vier alternative Marktplätze umfasste, wurden im Falle des *Google Play Stores* eine Infektionsrate von 0,02% festgestellt, bei den alternativen Marktplätzen reichte die Infektionsrate bereits von 0,2% bis 0,47%. Dementsprechend war die Infektionsrate in alternativen Marktplätzen um mehr als eine Größenordnung höher, als im offiziellen *Google Play Store*. Grund dafür ist laut den AutorInnen, dass Schadsoftware, die aus dem

Google Play Store bereits entfernt wurde, oftmals noch Monate später in alternativen Marktplätzen zu finden ist. Ein Beispiel hierfür ist die Schadsoftware *DroidDream*, welche im März 2011 innerhalb von 48 Stunden rund 260000 Android Endgeräte infizierte, bevor Google die Schadsoftware aus dem offiziellen Marktplatz entfernte. Diese war allerdings auch drei Monate später noch in alternativen Marktplätzen auffindbar.

In einer weiteren Studie zeigten (Zhou, Zhou u. a. 2012), dass 5% bis 13% aller Applikationen in sechs untersuchen alternativen Marktplätzen *repackaged* (siehe Kapitel 3.1.2) waren. Dabei werden meist legitime Applikationen aus dem offiziellen *Google Play Store* genommen, modifiziert und in alternativen Marktplätzen veröffentlicht.

Version	Codename	API	Distribution
2.2	Froyo	8	0.7%
2.3.3 - 2.3.7	Gingerbread	10	11.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	9.6%
4.1.x	Jelly Bean	16	25.1%
4.2.x		17	20.7%
4.3		18	8.0%
4.4	KitKat	19	24.5%

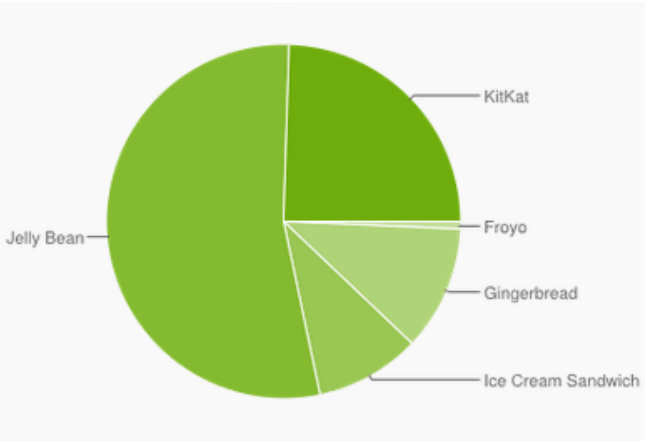


Abbildung 10: Android Versionen im Umlauf
(Quelle: https://developer.android.com/about/dashboards/index.html)

Abbildung 10 zeigt die Verteilung von Android Versionen am 9. September 2014. Versionen, die von weniger als 0,1% der EndbenutzerInnen verwendet werden, sowie alle Versionen unter 2.2 Froyo wurden nicht berücksichtigt.

Zu sehen ist, dass es immer noch viele BenutzerInnen gibt, denen wichtige Sicherheits-Features, wie der Warn-Dialog beim Versenden von Mehrwert SMS-Nachrichten ab Version 4.2 oder SELinux ab Version 4.3, nicht zur Verfügung stehen.

3.1.5 Datensätze

Die im Zuge dieser Arbeit verwendeten Android Schadsoftware Samples stammen aus dem Android Malware Genome Project (Zhou und Jiang 2012) sowie von Contagio Mobile[2]. Das Android Malware Genome Project entstand im Zuge der Arbeit von Zhou and Jiang und beinhaltet 1260 Schadsoftware Samples, die in 49 Familien eingeteilt werden können. Dieser Datensatz wurde zwischen August 2010 und Oktober 2011 aufgebaut. Contagio Mobile bietet eine *Dropbox*, von welcher Forscher Schadsoftware Samples beziehen oder

2. http://contagiominidump.blogspot.nl

hochladen können. Diese Datenbank beinhaltet rund 380 Schadsoftware Samples, die zwischen September 2012 und September 2014 gesammelt wurden.

Als weitere, und weit umfangreichere Quelle, die allerdings für diese Arbeit nicht Verwendung fand, könnte VirusTotal herangezogen werden. Bei VirusTotal, einer Tochtergesellschaft von Google, handelt es sich um einen Dienst zum Erkennen von Schadsoftware. Sampels, die an VirusTotal übermittelt werden, können von mehr als 50 verschiedenen Virenscannern untersucht werden. Durch Nutzung der privaten API[3] ist es möglich, als Schadsoftware erkannte Samples durch Angabe eines Hashes herunterzuladen.

3.2 Erkennung von Schadsoftware

Um Android Schadsoftware effizient erkennen zu können, wird von kommerziellen Anti-Malware Systemen meist auf die Erkennung von bestimmten Mustern, sogenannten Signaturen, gesetzt. Diese Signaturen enthalten charakteristische Muster, die eine Schadsoftware eindeutig identifizieren sollen.

Um diese Signaturen erstellen zu können, werden bekannte Android Schadsoftware Samples meist einer statischen Codeanalyse oder einer dynamischen Verhaltensanalyse unterzogen. Da beide Verfahren Vor- und Nachteile aufweisen, wird oftmals eine Kombination aus beiden Verfahren eingesetzt. Besonders die dynamische Analyse kann hierbei von einer vorangegangenen statischen Analyse profitieren. In den folgenden Unterkapiteln wird näher auf die statische Analyse und deren Probleme durch Quelltext-Verschleierung, auf die dynamische Verhaltensanalyse und Möglichkeiten eine solche Analyse zu erkennen, sowie auf kombinierte Verfahren eingegangen. Außerdem werden jeweils bestehende Arbeiten, beziehungsweise Frameworks, vorgestellt.

3.2.1 Statische Analyse

Bei der statischen Analyse wird eine Applikation analysiert, ohne diese tatsächlich auszuführen. Werkzeuge, die dafür verwendet werden, reichen von Dis-Assemblern, De-Compilern und komplexen Quellcode-Analyse-Programmen bis hin zu einfachen Werkzeugen wie beispielsweise `grep`. Der Vorteil von statischer Analyse liegt darin, dass sie den gesamten Quellcode abdeckt. Sie kann so auch zeigen, wie sich Applikationen unter ungewöhnlichen Bedingungen verhalten, da auch Teile des Quellcodes analysiert werden, die unter normalen Umständen niemals ausgeführt würden. Statische Analyse alleine kann allerdings nur ein ungefähres Bild liefern. Es ist damit nicht möglich, jedes Verhalten einer Applikation vollständig vorherzusagen (Farmer und Venema 2004).

Auch wenn die Werkzeuge für statische Analyse stetig weiterentwickelt werden, gibt es laut (Moser, Kruegel und Kirda 2007) ein fundamentales Limit, welcher Code noch untersucht werden kann und welcher nicht. Dieses Limit ergibt sich laut (Anderson 2008) dadurch, dass auch bei der statischen Analyse oft nur in der Theorie alle Zustände und

3. https://www.virustotal.com/de/documentation/private-api/

Ausführungs-Pfade simuliert werden können. Wenn eine Methode, die keinerlei Schleifen enthält, n Zustände aufweist, von denen es mehrere mögliche Übergänge in verschiedene Folge-Zustände gibt, müssten bereits bis zu 2^n Ausführungs-Pfade berücksichtigt werden. In der Praxis sind es weniger, da sich viele Pfade überschneiden. Wenn in dieser Methode andere Methoden-Aufrufe und deren mögliche Rückgabelwerte berücksichtigt werden, ist die Anzahl der möglichen Ausführungs-Pfade laut (Anderson 2008) bereits doppelt exponentiell. Werden auch noch Schleifen berücksichtigt, kann es unendlich viele mögliche Ausführungs-Pfade geben. Da es nicht möglich ist, so viele Pfade zu untersuchen, werden diese normalerweise limitiert. Schleifen werden beispielsweise nur eine bestimmte fixe Anzahl von Durchläufen untersucht oder beim Analysieren einer Methode wird ein oberes Limit von zu analysierenden Ausführungs-Pfaden gesetzt.

Wurde der Quellcode zu stark durch Verschleierungs-Techniken modifiziert, stößt statische Analyse schnell an ihre Grenzen, da es eben zu viele mögliche Ausführungs-Pfade gibt. Diese Grenzen sind von geringer Tragweite, wenn es um das Finden von Programmierfehlern geht, wofür statische Analyse normalerweise eingesetzt wird. Kommt es allerdings zu Schadsoftware, deren Quellcode absichtlich so konstruiert wurde, dass er einer statischen Analyse widersteht, stellt dass ein Problem dar. Aus diesem Grund kann statische Analyse alleine nicht sicherstellen, das Schadsoftware zuverlässig identifiziert wird. Statische Analyse sollte deshalb laut (Moser, Kruegel und Kirda 2007) durch dynamische Analyse ergänzt werden, da diese deutlich weniger anfällig für Quelltext-Verschleierungen und -Transformationen ist.

Quellcode-Verschleierung

Die *Code Obfuscation* oder Quellcode Verschleierung ist eine Technik, die es erschweren soll, Applikationen zu verstehen. Dafür wird eine neue, modifizierte Version einer Applikation generiert, die sich funktionell nicht unterscheidet. Ursprünglich wurden diese Techniken entwickelt, um das geistige Eigentum von Software-EntwicklerInnen zu schützen. Diese werden allerdings auch von EntwicklerInnen von Schadsoftware genutzt, um schädlichen Code so zu verstecken, dass er nicht erkannt werden kann (You und Yim 2010).

Verschleierung kann auf verschiedenen Ebenen stattfinden, beispielsweise auf Ebene des Quellcodes oder auf der des kompilierten Maschinencodes. Einige Verschleierungs-Techniken, sind laut (Moser, Kruegel und Kirda 2007) folgende:

- **Verschleierung von Konstanten:** Konstanten sind allgegenwärtig in der Programmierung. Im einfachsten Fall werden sie in ein Register geladen. Diese Ladeoperation kann durch eine Abfolge von Instruktionen, die semantisch äquivalent aber schwerer zu analysieren sind, ersetzt werden. Beispielsweise können Konstanten kompliziert berechnet werden, anstatt sie einfach zuzuweisen.
- **Verschleierung der Kontrollstruktur:** Eine Grundvoraussetzung, um fortgeschrittene statische Analysen durchzuführen, ist die Erstellung von Kontrollstruktur-Graphen. Ein Kontrollstruktur-Graph oder *Control Flow Graph* ist ein gerichteter

Graph $G = (V, E)$, dessen Knoten $u, v \in V$ Blöcke repräsentieren, während die Kanten $e \in E : u \rightarrow v$ einen möglichen Kontrollfluss von u nach v darstellen. Ein Block besteht hierbei aus einer Sequenz von Instruktionen ohne Sprunganweisungen. `JMP`- und `CALL`-Instruktionen stellen Kanten dar. Die Idee hinter der Verschleierung von Kontrollstrukturen ist, diese Sprunganweisungen durch andere Instruktionen zu ersetzen, so dass der Kontrollfluss nicht beeinflusst wird, es allerdings schwierig ist, die jeweiligen Ziele durch statische Analyse herauszufinden. Dafür kann die Verschleierung von Konstanten, in diesem Fall einer Speicheradresse, eingesetzt werden.

- **Verschleierung von Daten:** Der Speicherort von Daten wird oftmals durch Konstanten oder mittels konstanter Offsets relativ zu einem bestimmten Register festgelegt. In beiden Fällen kann wiederum der genaue Speicherort oder der Zugriff auf bestimmte Daten durch die Nutzung verschleierter Konstanten versteckt werden.

Techniken zur Mutation von Schadsoftware

Auf Ebene des Maschinencodes, gibt es laut (You und Yim 2010) verschiedene einfache Techniken, um neue Mutationen oder Varianten von Schadsoftware zu erstellen. Dadurch soll deren Auffindbarkeit durch Signaturen erschwert werden.

- **Einfügen von totem Code:** Das Einfügen von totem Maschinencode, der nicht ausgeführt wird, ist eine einfache Technik, um das Erscheinungsbild einer Applikation zu verändern, ihre Funktionalität allerdings unverändert zu lassen. Ein Beispiel hierfür wäre das willkürliche Einfügen des `NOP` Kommandos (NOP steht für *No Operation* und hat keine Auswirkung auf den Programmablauf). Eine auf Signaturen basierende Erkennung ist möglich, wenn vor der Analyse ineffektive Befehle entfernt werden.

- **Neuzuordnung von Registern:** Eine weitere Technik ist das Vertauschen von Registern. Von Generation zu Generation werden Register zufällig vertauscht, ohne das Verhalten der Applikation zu verändern. Dieser Technik kann durch die Nutzung von *Wildcards* in den Signaturen gut entgegengewirkt werden.

- **Umsortieren von Subroutinen:** Auch durch das zufällige Umsortieren von Subroutinen können neue Schadsoftware Generationen entstehen. Einzelne Subroutinen können allerdings immer noch durch Signaturen erkannt werden.

- **Substitution von Instruktionen:** Bei der Substitution von Instruktionen werden Instruktionen durch gleichwertige ersetzt. So kann zum Beispiel ein `XOR` durch `SUB` oder `MOV` durch `PUSH/POP` ersetzt werden. Auch dieser Technik kann durch die Nutzung von *Wildcards* in den Signaturen entgegengewirkt werden.

- **Transposition von Code:** Bei der Transposition von Maschinencode werden Instruktionen zufällig umsortiert, während das Verhalten der Applikation erhalten bleibt. Das kann auf zwei Arten bewerkstelligt werden. Entweder werden `JMP` Instruktionen benutzt, um die Ausführungsreihenfolge zu erhalten oder es werden nur

Blöcke mit Instruktionen, die keinen Einfluss auf andere Instruktionen haben, verschoben. Diese zu finden ist allerdings aufwendig.

(Automatische) Mutation von Schadsoftware

Um die Erkennung zu erschweren, kann fortgeschrittene Schadsoftware selbst mutieren. Laut (Ször und Ferrie 2001) können vier verschiedene Arten von Mutation unterschieden werden, wobei jede Stufe eine größere Herausforderung darstellt, um sie zu identifizieren:

1. **Verschlüsselt:** Bei der ersten Variante wird Verschlüsselung eingesetzt, um einer Erkennung zu entgehen. Dafür besteht verschlüsselte Schadsoftware aus einem Verschlüsselungsmodul und dem verschlüsselten Hauptteil. Das Verschlüsselungsmodul entschlüsselt den Hauptteil immer dann, wenn die infizierte Applikation ausgeführt wird. Da bei jeder Infektion ein zufälliger Schlüssel verwendet wird, ist der verschlüsselte Hauptteil der Applikation einzigartig und kann nur schwer durch Signaturen erkannt werden. Die Erkennung solcher Schadsoftware ist allerdings möglich, ohne den verschlüsselten Hauptteil zu untersuchen. In den meisten Fällen ist das Verschlüsselungsmodul einzigartig genug, um erkannt zu werden.

2. **Oligomorph:** Bei oligomorpher Schadsoftware mutiert auch das Verschlüsselungsmodul von einer Generation zur nächsten. Die Anzahl von mögliche Mutationen ist allerdings begrenzt, wodurch auch oligomorphe Schadsoftware durch Signaturen erkannt werden kann.

3. **Polymorph:** Polymorphe Schadsoftware kann, mit Hilfe von Mutations-Techniken, eine theoretisch endlose Anzahl von neuen Verschlüsselungsmodulen erzeugen, die wiederum verschiedene Methoden zum Entschlüsseln des Hauptteiles unterstützen können. Identifiziert werden kann eine solche Schadsoftware in manchen Fällen durch unzulänglich generierte Zufallszahlen in der Mutations-Engine. Anderenfalls kann polymorphe Schadsoftware nur durch dynamische Analyse erkannt werden.

4. **Metamorph:** Metamorphe Schadsoftware verzichtet auf ein Verschlüsselungsmodul. Stattdessen werden wiederum Mutations-Techniken eingesetzt, woraufhin sich die Schadsoftware selbst neu kompiliert, um eine neue Generation zu erzeugen. Da dadurch jedes Mal eine komplett neue Applikation entsteht, kann diese auch meist nur durch dynamische Verhaltensanalyse erkannt werden. Für Android ist diese Art von Schadsoftware allerdings weitgehend uninteressant, da für die Generierung neuer Generationen auf dem infizierten System eine Umgebung zum Kompilieren von Applikationen vorhanden sein muss.

Vor- und Nachteile von statischer Analyse

+ Deckt den gesamten Quellcode ab.

+ Es wird auch Code berücksichtigt, der unter normalen Umständen nicht ausgeführt würde.

+ Im Gegensatz zur dynamischen Analyse sehr schnell (wenn automatisiert).
+ Verfahren zur Erkennung von dynamischer Analyse können erkannt werden.
− Es besteht ein fundamentales Limit bei der Anzahl von Ausführungs-Pfaden, die berücksichtigt werden können.
− Quellcode Verschleierung und Mutation von Software können die Anzahl von möglichen Ausführungs-Pfaden enorm erhöhen.

Bisherige Arbeiten

Trotz der Probleme, die statische Analyse mit sich bringen kann, wurde in den vergangenen Jahren durchaus erfolgreiche Forschung in diese Richtung betrieben:

- **ReadElf:** Im Jahr 2009 versuchten (Schmidt u. a. 2009) mit Hilfe des `readelf` Werkzeugs, Funktionsaufrufe von Android Applikationen zu extrahieren. Die extrahierten Funktionsaufrufe wurden anschließend mittels *PART*, *Prism* und *Nearest Neighbor* Algorithmen analysiert und mit bekannter Schadsoftware verglichen.
- **SOM:** Im Jahr 2010 präsentierten (Barrera u. a. 2010) eine empirische Analyse des Android Berechtigungs-Systems mit Hilfe von SOM. Dazu wurden 1100 Applikationen untersucht. Mit Hilfe des SOM Algorithmus wurde gezeigt, wie bestimmte Berechtigungen von EntwicklerInnen genutzt werden. Es wurde herausgefunden, dass Android zwar über eine große Anzahl von verschiedenen Berechtigungen verfügt, allerdings nur eine geringe Anzahl davon wirklich aktiv von EntwicklerInnen genutzt wird. Es wurden dabei Berechtigungen entdeckt, die sehr weitläufig sind und so eine große Anzahl an Features abdecken. Andere Berechtigungen sind wiederum nur für ein bestimmtes Feature verantwortlich. Die durchgeführte empirische Analyse sollte eine Basis für mögliche Verbesserungen beim Android Berechtigungs-System liefern.
- **DED:** Im Jahr 2011 stellte (Enck u. a. 2011) den *DED* De-Compiler vor. Dabei handelt es sich um einen Dalvik De-Compiler, der in der Lage ist, Java Quellcode aus einer APK Installationsdatei zu rekonstruieren. Dies geschieht, indem Dalvik Bytecode in Java Bytecode umgewandelt wird und anschließend verlorene Datentypen sowie Klassen und Methoden wiederhergestellt werden. Laut der AutorInnen existierte vor *DED* kein Werkzeug, das mit Dalvik Bytecode arbeiten konnte. Aufgrund des großen Unterschiedes zwischen Java Bytecode und Dalvik Bytecode war es nicht möglich, existierende Werkzeuge anzupassen. Mit Hilfe des entwickelten *DED* De-Compilers wurde eine Studie mit 1100 verbreiteten Android Applikationen durchgeführt. Es wurde festgestellt, dass sensible Informationen, besonders Identifizierungs-Merkmale von Endgeräten wie International Mobile Equipment Identity (IMEI) oder International Mobile Subscriber Identity (IMSI) und Positionsdaten missbraucht wurden. Schadsoftware konnte dabei keine identifiziert werden. Allerdings wurde gezeigt, dass bei der Entwicklung von Android Applikationen von EntwicklerInnen

wenig auf die sichere Verwendung von APIs und somit auf den Schutz sensibler Daten geachtet wird. Auch enthielten 51% der analysierten Applikationen Bibliotheken von Werbe- oder Analysenetzwerken. Viele Applikationen enthielten sogar mehr als eine Werbebibliothek.

- **Androguard:** Im selben Jahr wurde *Androguard* als Framework für statische Analyse vorgestellt. Dabei wird mit Hilfe von *DED* aus Android Applikationen Java Quellcode generiert. Unter anderem ist es mit *Androguard* möglich, Schadsoftware mit Hilfe von Signaturen zu identifizieren, die Ähnlichkeit zweier Anwendungen zu berechnen um *Repackaging* zu erkennen, und eine Vielzahl von Informationen aus einer Applikation auszulesen, beispielsweise welche Komponenten sie enthält. Aufgrund seiner Flexibilität wird *Androguard* bei vielen, auch dynamischen, Analyse-Frameworks als Basis verwendet (Desnos und Gueguen 2011).

- **RiskRanker:** Im Jahr 2012 schlugen (Grace u. a. 2012) ein Framework namens *RiskRanker* vor. Mit dessen Hilfe soll es möglich sein, *zero-day* Schadsoftware ausfindig zu machen. Das automatisierte System ist in der Lage zu erkennen, ob eine Applikation gefährliche Verhaltensmuster aufweist. Beispiele hierfür sind die Anwendung von *Root-Exploits* oder das Versenden von Mehrwert SMS-Nachrichten im Hintergrund. Mit Hilfe der gesammelten Daten wird anschließend eine priorisierte Liste von Applikationen erstellt, die einer genaueren Untersuchung bedürfen. In der durchgeführten Studie wurden insgesamt 118318 Applikationen aus verschiedenen Marktplätzen untersucht. Das System fand darunter 3281 Applikationen, die als gefährlich eingestuft wurden. Diese 3281 Applikationen enthielten 718 bekannte Schadsoftware Samples aus 29 Schadsoftware-Familien, wovon es sich bei 322 um *zero-day* Schadsoftware (11 Familien) handelte.

3.2.2 Dynamische Analyse

Bei der dynamischen Analyse oder auch Verhaltensanalyse wird eine Applikation während deren Ausführung überwacht, beziehungsweise analysiert. Werkzeuge, die dafür Verwendung finden, sind Debugger, Tracer, Emulatoren und Netzwerk Sniffer. Der Vorteil von dynamischer gegenüber statischer Analyse ist, dass verschleierter Code keine Probleme macht, da nicht der Quellcode direkt analysiert wird, sondern das Verhalten der Applikation. Nachteile sind die im Vergleich zur statischen Analyse oftmals langen Ausführungszeiten und dass nicht immer alle Pfade erreicht werden. Es werden dabei nur diejenigen Pfade analysiert, die auch tatsächlich ausgeführt werden. Da es bei komplexen Applikationen nur schwer möglich ist, alle vorhandenen Pfade zur Ausführung zu bringen, ist auch die dynamische Analyse nicht zu 100% zuverlässig (Farmer und Venema 2004).

Aus diesem Grund wird oftmals vor der dynamischen Analyse einer Android Applikation eine statische Analyse durchgeführt, um beispielsweise alle vorhandenen Komponenten im Vorfeld zu kennen. Da jede Komponente unter Android einen Einstiegspunkt in die Applikation darstellt, kann in der dynamischen Analyse so zum Beispiel versucht wer-

den, jede Komponente einzeln auszuführen, um eine größere Abdeckung der möglichen Ausführungspfade zu erreichen.

Neben der Tatsache, dass es nur schwer möglich ist, bei komplexeren Applikationen alle Ausführungspfade abzudecken, wird dies laut (Paleari u. a. 2009) im Fall von Schadsoftware oftmals zusätzlich erschwert. Umso länger es dauert, neue Schadsoftware zu identifizieren, umso länger *überlebt* sie (zum Beispiel im offiziellen *Google Play Store*) und umso mehr Endgeräte können befallen werden. Da sich AutorInnen von Schadsoftware darüber im Klaren sind, dass diese einer dynamischen Analyse, im Falle vom *Google Play Store* durch ein Framework namens *Bouncer*, unterzogen wird, versuchen sie durch Tests festzustellen, ob die Applikation auf einem echten Endgerät oder in einem Emulator, einer Virtuellen Maschine oder einem Debugger ausgeführt wird. Wenn ein solcher Test ergibt, dass die Applikation in einer solchen Umgebung läuft, wird versucht, das schädliche Verhalten zu verbergen.

Hypervisoren und Emulatoren

Unter einem Hypervisor (auch Virtual Machine Monitor (VMM) genannt) oder einem Emulator werden Programme verstanden, die Hardware simulieren können. In dieser simulierten Umgebung können ganze Betriebssysteme so ausgeführt werden, als ob sie auf echter Hardware laufen würden (Raffetseder, Kruegel und Kirda 2007).

In (Popek und Goldberg 1974) werden VMs als ein effizientes, isoliertes Duplikat einer echten Maschine beschrieben. Außerdem muss ein VMM folgende Charakteristiken erfüllen:

- Der VMM stellt Programmen eine Umgebung zur Verfügung, die außer der Verfügbarkeit von System-Ressourcen der einer echten Maschine gleicht.
- Programme, die in einer solchen Umgebung ausgeführt werden, zeigen höchstens einen geringen Geschwindigkeitsunterschied. Ein signifikanter Anteil von Instruktionen muss dafür auf der echten Hardware ausgeführt werden.
- Der VMM behält die komplette Kontrolle über die System-Ressourcen. Programme, die in einer VM ausgeführt werden können nur auf Ressourcen zugreifen, die ihnen explizit zugewiesen wurden.

Der zweite Punkt unterscheidet einen VMM von einem Emulator. Emulatoren führen keinen Code direkt auf der Hardware aus, wodurch es oft zu großen Geschwindigkeitseinbußen kommt. Ein Emulator simuliert das Verhalten der CPU mit Hilfe von Software. Alle Instruktionen werden vom Emulator abgefangen, für die Ziel Plattform übersetzt und erst dann auf der Hardware ausgeführt. Ein Emulator ist deshalb, im Gegensatz zu einem VMM, in der Lage, auch Software, die für eine andere Prozessor-Architektur als die des Host-Computers kompiliert wurde, auszuführen (Raffetseder, Kruegel und Kirda 2007).

Erkennung von virtuellen Umgebungen

Wie bereits zuvor beschrieben, sind laut (Raffetseder, Kruegel und Kirda 2007) EntwicklerInnen in der Lage zu erkennen, ob ihre Software in einem Emulator oder einer VM ausgeführt wird. Dafür gibt es verschiedene Möglichkeiten, unter anderem folgende:

- **Unterschiede im Verhalten:** Die Ausführung einer Applikation auf echter Hardware und in einer virtuellen Umgebung sollte zum selben Ergebnis führen, um eine perfekte Emulation zu gewährleisten. Gibt es Unterschiede, kann ein Emulator erkannt werden. Besonderes Augenmerk wird hierbei auf Fehler von CPUs und modellspezifische Register gelegt. Moderne CPUs sind sehr komplex, wodurch jede Prozessor-Familie Fehler enthält. Durch das Vorhandensein oder Fehlen von bestimmten Fehlern kann eine CPU eindeutig erkannt werden.
- **Unterschiede im Timing:** Die Performance einer Applikation, die in einer virtuellen Umgebung ausgeführt wird, ist immer geringer als wenn sie auf echter Hardware ausgeführt würde. Zu messen, ob die absolute Performance einer bestimmten Applikation zur verwendeten CPU passt, um Virtualisierung festzustellen, ist allerdings sehr fehleranfällig. Stattdessen wird die relative Performance zwischen zwei Instruktionen gemessen. Werden die beteiligten Instruktionen richtig gewählt, kann wiederum eine bestimmte CPU Familie oder eben ein VMM beziehungsweise Emulator erkannt werden.
- **Spezifische Hardware:** In einer simulierten Umgebung müssen alle peripheren Geräte, beispielsweise Grafikkarten oder Festplatten, emuliert werden. Bestimmte Emulatoren implementieren diese virtuellen Geräte auf eine charakteristische Art und Weise, die erkannt werden kann.

Um solche hardwarespezifischen Unterschiede (sogenannte *red-pills*) zu finden, die eine CPU oder einen Emulator eindeutig identifizieren können, wurde von (Paleari u. a. 2009) eine systematische Technik vorgeschlagen. Damit war es den ForscherInnen möglich, tausende dieser *red-pills* für zwei weit verbreitete Emulatoren, QEMU[4] und BOCHS[5], zu generieren. Laut der AutorInnen lässt sich diese Technik auch einfach auf andere virtuelle Umgebungen, beispielsweise die von VirtualBox[6] oder VMware[7] anwenden. Für QEMU, wobei es sich um den Standart-Emulator von Android handelt, der mit jedem SDK ausgeliefert wird und auch in *Bouncer* Verwendung findet, konnten 20729 solcher eindeutigen *red-pills* identifiziert werden.

Ob solche *red-pills* in Schadsoftware zum Einsatz kommen, kann oftmals nur durch statische Analyse erkannt werden.

4. http://www.qemu.org
5. http://bochs.sourceforge.net
6. http://www.virtualbox.org
7. http://www.vmware.com

Vor- und Nachteile von dynamischer Analyse

+ Dynamische Analyse ist resistent gegen Quellcode-Verschleierung.
+ Dynamische Analyse berücksichtigt auch Code, der beispielsweise aus dem Internet nachgeladen wird.
− Dynamische Analyse ist meist ressourcenintensiver als statische Analyse.
− Dynamische Analyse kann erkannt werden.
− Es wird eine isolierte Sandbox benötigt, deren Ausgangszustand exakt wiederherstellbar ist.
− Die zu analysierende Applikation muss ausgeführt werden, was sehr zeitintensiv sein kann, um eine gute Codeabdekung zu erreichen.
− Es ist nur schwer möglich, den gesamten Code abzudecken.

Bisherige Arbeiten

Vor allem im Bereich der dynamischen Verhaltens-Analyse von Android Applikationen wurde in den vergangenen Jahren geforscht, woraus einige Analyse Frameworks hervorgegangen sind:

- **TaintDroid:** Im Jahr 2010 präsentierten (Enck u. a. 2010) *TaintDroid*. Dabei handelte es sich ursprünglich um eine modifizierte Android 2.1 Version. *TaintDroid* steht mittlerweile außerdem für die Versionen 2.3, 4.1 und 4.3 von Android zur Verfügung[8] und wird in vielen dynamischen Analyse Frameworks eingesetzt. Es existierte allerdings zum Zeitpunkt des Schreibens dieser Arbeit noch kein Release für das aktuelle Android 4.4 (KitKat). Das Ziel von *TaintDroid* ist es, den Fluss von sensiblen Daten durch nicht vertrauenswürdige Applikationen zur Laufzeit zu verfolgen und dabei *privacy leaks* zu erkennen. Da *TaintDroid* in der Dalvik VM implementiert wurde, ist es damit nicht möglich, nativen Code zu überwachen.
- **CrowDroid:** Im Jahr 2011 veröffentlichten (Burguera, Zurutuza und Nadjm-Tehrani 2011) eine *Crowdsourcing* Applikation namens *CrowDroid*. Dabei handelt es sich um eine Anwendung, die mittels `strace`[9] Systemaufrufe von installierten Applikationen zur Laufzeit aufzeichnet. Diese wurden anschließend zur Analyse an einen zentralen Server gesendet, welcher mittels ML Techniken entschied, ob die ausgeführte Applikation gut- oder bösartig ist.

8. http://appanalysis.org/download.html
9. http://sourceforge.net/projects/strace

- **DroidBox:** Im selben Jahr wurde (Lantz 2014) *DroidBox*[10] präsentiert. Dieses Framework wurde als Teil des *Google Summer of Code (GSoC) 2011* entwickelt. *DroidBox* verbindet *TaintDroid* mit einigen Modifikationen der Android System-Bibliotheken (siehe Kapitel 2.2.2). Es werden zusätzlich das Lesen und Schreiben von Dateien, Aktivitäten der Kryptographie API, Netzwerkverbindungen sowie Netzwerkverkehr und das Versenden von SMS-Nachrichten sowie Telefonanrufe überwacht.

- **DroidScope:** Im Jahr 2012 schlugen (Yan und Yin 2012) ein Framework namens *DroidScope* vor. Sie modifizierten QEMU, sowie die Dalvik VM dahingehend, dass detaillierte native und Dalvik *Instruction Traces* aufgezeichnet wurden. Außerdem wurden Android API Aufrufe und, ähnlich wie unter *TaintDroid*, *privacy leaks* überwacht.

- **CopperDroid:** Im Jahr 2013 wurde von (Reina, Fattori und Cavallaro 2013) *CopperDroid* veröffentlicht. Auch dieses Framework verwendet QEMU als Emulator. Dieser wurde wie bei *DroidScope* so modifiziert, dass Systemaufrufe aufgezeichnet werden. Der Fokus dieser Arbeit lag allerdings nicht darauf, ein Framework zu entwickeln, sondern zu zeigen, wie durch eine Analyse auf Basis von Systemaufrufen Schadsoftware identifiziert werden kann.

3.2.3 Kombinierte Analyse

Wie bereits in Kapitel 3.2.1 und Kapitel 3.2.2 beschrieben wurde, ist es sinnvoll, statische Analyse in Kombination mit einer dynamischen Verhaltensanalyse einzusetzen. Bei der statischen Analyse können beispielsweise Informationen gesammelt werden, die hilfreich sind, um bei der nachfolgenden dynamischen Analyse eine möglichst gute Codeabdeckung zu erreichen. Viele dynamische Analyse Frameworks machen deshalb Gebrauch von einer vorangegangenen statischen Analyse, darunter:

- **AASandbox:** Im Jahr 2010 erschien das erste dynamische Analyse Framework namens *AASandbox* (Blasing u. a. 2010). Das Framework führt zuerst eine statische Analyse durch, um nach verdächtigen Mustern zu suchen. Danach wird die zu untersuchende Applikation in einer isolierten Umgebung (Sandbox) ausgeführt, um Systemaufrufe mit Hilfe eines Kernel-Moduls aufzuzeichnen, die später analysiert werden. Die *AASandbox* und der dazugehörige Erkennungs-Algorithmus wurden so implementiert, dass sie in der *Cloud* ausgeführt werden können. Leider existierten zu dieser Zeit keine umfangreichen Android Schadsoftware Datensätze, weshalb nur mit den damaligen Top 150 Applikationen aus dem *Google Play Store* und einem selbst entwickelten *Fork-Bomb* Sample experimentiert wurde.

10. http://code.google.com/p/droidbox/

- **Structured Control Flow:** Im selben Jahr veröffentlichten (Cesare und Xiang 2010) ein Framework, das mit Hilfe von Kontrollstrukturen Schadsoftware erkennen sollte. Dafür wurde zuerst durch eine Entropie-Analyse festgestellt, ob es sich um gepackte Schadsoftware handelt. Unter gepackt wird in diesem Zusammenhang verstanden, dass der Schadcode in komprimierter oder verschlüsselter Form vorliegt und erst zur Laufzeit wiederhergestellt wird (siehe Kapitel 3.2.1). Handelt es sich um gepackte Schadsoftware, wird diese dynamisch ausgeführt, um versteckten Code aufzudecken. Es werden Kontrollstrukturen extrahiert, die als Signaturen dienen. Diese können mit einer Datenbank bekannter Schadsoftware abgeglichen werden. Um Schadsoftware zu klassifizieren, wird die Bearbeitungs-Distanz zwischen verschiedenen Kontrollstrukturen herangezogen. Auch dieses System wurde nur mit synthetischer Schadsoftware getestet, da zu diesem Zeitpunkt keine umfangreiche Datenbank mit Android Schadsoftware existierte.

- **Andrubis:** Im Jahr 2012 wurde *Andrubis* vom *International Secure Systems Lab* vorgestellt (Lindorfer 2012). Dabei handelte es sich um eine Erweiterung von Anubis[11] und die erste Analyse Plattform, bei der es BenutzerInnen möglich war, Android Applikationen über ein Web-Interface hochzuladen und analysieren zu lassen. *Andrubis* generiert nach Abschluss der Analyse einen Report, der die Ergebnisse der statischen und dynamischen Analyse enthält. Das erste Release von *Andrubis* baute auf *DroidBox* auf und unterstützte Android 2.1. *Andrubis* wurde seither stark weiterentwickelt und hat seit 2012 rund 900000 Android Applikationen (3500 pro Tag) analysiert (Weichselbaum u. a. 2014).

- **DroidRanger:** Im selben Jahr wurde *DroidRanger* (Zhou, Wang u. a. 2012) vorgestellt. Dabei werden Applikationen aufgrund der in ihrer Manifest-Datei deklarierten Berechtigungen vor-gefiltert, um nur Applikationen genauer zu untersuchen, die Berechtigungen anfordern, welche oftmals von Schadsoftware missbraucht werden. Die übrigen Applikationen werden einer dynamischen Verhaltensanalyse unterzogen, um bereits bekannte Schadsoftware zu identifizieren. Dafür wird ein Kernel-Modul verwendet, das Systemaufrufe aufzeichnet, die in bekannten Android *Exploits* Verwendung finden. Danach wird noch ein heuristisches Filterungs-Schema angewandt, um *zero-day* Schadsoftware zu finden. Bei der durchgeführten Analyse von 204020 Android Applikationen wurden 211 Schadsoftware Samples gefunden. Durch das heuristische Filtern konnten darin 40 *zero-day* Samples aus 2 Familien identifiziert werden, 11 davon im offiziellen *Google Play Store*.

- **AppsPlayground:** Im Jahr 2013 veröffentlichten (Rastogi, Chen und Enck 2013) *AppsPlayground*. Dieses Framework baut auf *TaintDroid* auf, um *privacy leaks* zu überwachen. Auch hier werden zusätzlich verschiedene API- und Systemaufrufe aufgezeichnet, während Android mittels QEMU emuliert wird. BenutzerInnen-Eingaben werden durch ein zuvor für jede Applikation definiertes Modell simuliert. Dieses Modell wird im Vorfeld aus Informationen, die aus GUI-Elementen extrahiert werden,

11. https://anubis.iseclab.org

generiert. Damit konnte im Durchschnitt eine 30% höhere Codeabdeckung als durch Zufallseingaben bewirkt werden.

- **Mobile-Sandbox:** Im Jahr 2013 wurde die *Mobile-Sandbox* von (Spreitzenbarth u. a. 2013) veröffentlicht. Sie zeigt starke Ähnlichkeit zu *Andrubis*, da auch sie auf *DroidBox* aufbaut und *TaintDroid* zum Identifizieren von *privacy leaks* verwendet. Um Systemaufrufe aufzuzeichnen, wurde allerdings nicht der Emulator erweitert, sondern `ltrace`[12] eingesetzt.

3.3 Automatische Generierung von Input

Bei der dynamischen Analyse von Android Applikationen sollte eine möglichst gute Codeabdeckung gewährleistet werden. Dafür können Eingaben manuell von BenutzerInnen erfolgen, wie es zum Beispiel bei *CrowDroid* der Fall ist. Werden die Daten für die Analyse allerdings nicht durch *Crowdsourcing* generiert, sondern versucht, eine große Anzahl von Applikationen automatisiert zu analysieren, müssen normalerweise Eingaben und Events automatisch generiert werden.

Google stellt für diesen Zweck als Teil des Android SDKs ein Werkzeug namens *Monkey* zur Verfügung. *Monkey* ist ein Programm, das im Emulator oder auf einem echten Endgerät ausgeführt werden kann, um pseudo-zufällige Streams von BenutzerInnen-Events zu erzeugen. Diese enthalten unter anderem Klick-, Touch- und Swipe-Events sowie einige System-Events (Google 2014k).

Eine bessere Codeabdeckung kann allerdings erzielt werden, wenn diese Events nicht zufällig generiert werden. Wie bereits beschrieben, konnte *AppsPlayground* eine 30% höhere Codeabdeckung erzielen, indem zuvor GUI-Elemente untersucht wurden, um daraus ein Modell für die Generierung von Eingaben zu erstellen. Da GUI-Elemente unter Android normalerweise in Form von XML-Dateien vorliegen, können diese relativ einfach aus einer APK extrahiert werden.

Auch (Amalfitano u. a. 2012) versuchten, Android Applikationen mit Hilfe ihrer entwickelten *AndroidRipper* Software in einer strukturierten Form auszuführen. *AndroidRipper* analysiert dafür dynamisch die GUI einer Applikation um passende Event-Sequenzen zu generieren. Es hat sich allerdings gezeigt, dass dieses Verfahren besser dafür geeignet ist, Fehler in Applikationen zu finden als damit eine dynamische Schadsoftware-Analyse durchzuführen.

Laut (Elish, Yao und Ryder 2012) wird schädlicher Code ohnehin nur in wenigen Fällen durch GUI-Events zur Ausführung gebracht. Viel öfter verlässt sich Schadsoftware auf System-Events wie zum Beispiel auf die Android Broadcasts.

12. http://www.ltrace.org

3.4 Maschinelles Lernen

Unter Maschinellem Lernen (ML) kann das automatische Finden von Mustern in Daten durch Algorithmen verstanden werden. Mit Hilfe dieser Muster ist es beispielsweise möglich, Daten zu klassifizieren (Bishop 2006). Eine formellere Definition von ML durch (Mitchell 1997) lautet folgendermaßen:

> "... A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E ..." (2)

ML wird oftmals mit *Data Mining* verwechselt, da sich beide Disziplinen stark überschneiden. Im Unterschied zu ML, konzentriert sich das *Data Mining* allerdings auf das Finden von neuen, zuvor unbekannten Informationen oder Eigenschaften in Daten.

Die Anwendungsbereiche von ML sind vielfältig und enthalten im Grunde alle Bereiche, in denen mit hochdimensionalen Daten gearbeitet wird.

- Zeichenerkennung und Erkennung von Handschrift
- Gesichtserkennung
- Geräuscherkennung
- Verständnis gesprochener Sprache
- Vorhersage von Börsenkursen
- Wetter- und Klima-Prognosen
- Medizin, beispielsweise Genom-Entschlüsselung
- Verhaltensanalyse

Abbildung 11 zeigt handgeschriebene Ziffern. Jede dieser Ziffern entspricht einer 28x28 Pixel großen Bilddatei und kann durch einen Vektor $X_s = (x_{s1}, x_{s2}, ..., x_{s784})$, der 784 reelle Zahlen enthält, beschrieben werden. Das Ziel ist es, eine Maschine zu bauen, die einen solchen Vektor X_s als Eingabe akzeptiert und als Ausgabe eine Zahl $0, ..., 9$ liefert. Das stellt, aufgrund der großen Variabilität von Handschriften, ein nicht triviales Problem dar. Dieses könnte durch manuell aufgestellte Regeln oder Heuristiken, die auf der Form der Striche basieren, gelöst werden. Bei dieser Herangehensweise würde es allerdings zu einer Vielzahl von Ausnahmen und Grenzfällen kommen, was zu schlechten Resultaten führen würde (Bishop 2006).

Bessere Resultate können durch einen ML-Ansatz erzielt werden, bei dem eine große Anzahl von N Zahlen $\{x_1, ..., x_N\}$, genannt Trainings-Set, genutzt wird, um ein adaptives Modell zu generieren. Die Kategorien der einzelnen Ziffern im Trainings-Set sind dabei

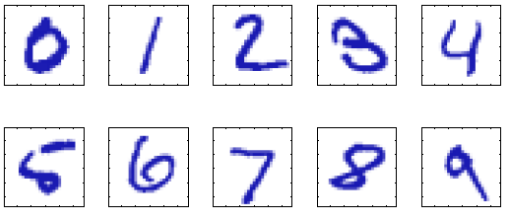


Abbildung 11: Handgeschriebene Ziffern (Bishop 2006)

bereits bekannt und werden den Ziel-Vektoren $T_s = (t_{s1}, t_{s2}, ..., t_{s784})$ zugewiesen. Für jedes Ziffern-Bild X_s wird dabei ein solcher Ziel-Vektor T_s erstellt (Bishop 2006).

Das Resultat des ML-Algorithmus kann als Funktion $y(x)$ ausgedrückt werden, welche ein neues Ziffern-Bild X_s als Eingabe erwartet und als Ergebnis einen Vektor $Y_s = (y_{s1}, y_{s2}, ..., y_{s784})$ liefert, der auf dieselbe Art encodiert ist wie die Ziel-Vektoren. Wie die Funktion $y(x)$ genau aussieht, wird während der Trainings-Phase, auch Lern-Phase genannt, auf Basis der Trainings-Daten bestimmt. Nachdem das Modell trainiert wurde, können damit neue Ziffern-Bilder aus einem Test-Set kategorisiert werden. Die Fähigkeit, auch Ziffern-Bilder richtig zu kategorisieren, die von denen aus dem Trainings-Set abweichen, nennt man *Generalisierung*. Das Trainings-Set enthält normalerweise nur einen Bruchteil aller möglichen Vektoren, wodurch Generalisierung eine wichtige Rolle im ML einnimmt (Bishop 2006).

Die Eingabe-Objekte werden typischerweise vor-verarbeitet, um das zu lösende Problem zu vereinfachen. Beim Ziffern-Problem können die Ziffern-Bilder beispielsweise so zugeschnitten und skaliert werden, dass sie eine Box mit fixer Größe ausfüllen. Das alleine kann die Variabilität der einzelnen Ziffern-Klassen bereits enorm reduzieren und so den Rechenaufwand minimieren. Alle Daten aus dem Trainings- und Test-Set müssen allerdings derselben Vorverarbeitung unterzogen werden (Bishop 2006).

Überwachtes Lernen

Beim überwachten Lernen oder *supervised learning* ist für alle Daten im Trainings-Set bereits im Vorfeld bekannt, welchem Ziel-Vektor beziehungsweise welcher Kategorie oder Klasse diese entsprechen. Ein Teilgebiet des überwachten Lernens ist die automatische Klassifizierung, bei der zuvor unbekannten Eingabe-Vektoren eine aus einer endlichen Anzahl diskreter Kategorien beziehungsweise Klassen, zugewiesen wird (Bishop 2006).

Nicht überwachtes Lernen

Im Gegensatz zum überwachten Lernen sind beim nicht überwachten Lernen oder *unsupervised learning* die Ziel-Vektoren nicht im Vorfeld bekannt. Ziele des nicht überwachten Lernens sind beispielsweise ähnliche, oder gleiche Datensätze zu finden (*Clustering*), herauszufinden, wie sich die Daten verteilen (*Density Estimation*) oder das Projizieren von hochdimensionalen Daten hinunter in die zweite oder dritte Dimension, um sie zu visualisieren (Bishop 2006).

Bestärkendes Lernen

Unter bestärkendem Lernen oder *reinforcement learning* wird das Finden von geeigneten Aktionen in bestimmten Situationen verstanden, um den Gewinn zu maximieren. Auch hier verfügt der Lernalgorithmus, wie beim nicht überwachten Lernen, über keine Beispiele, wie die optimale Lösung auszusehen hat. Der Algorithmus muss sich hierbei an die optimale Lösung durch Versuch und Irrtum herantasten (Bishop 2006).

3.5 Künstliche Neuronale Netze

In dieser Arbeit sollen schädliche Android Applikationen mit Hilfe von SOM beziehungsweise CP-ANN identifiziert werden. Bei beiden Modellen handelt es sich um ein künstliches neuronales Netz (ANN), wobei CP-ANN auf SOM aufbaut und im Gegensatz zu SOM überwachtes Lernen ermöglicht.

Der Begriff „neuronales Netz“ wurde bereits in den 1940er Jahren geprägt und hat seine Wurzeln im Versuch, die Informationsverarbeitung in biologischen Systemen mathematisch zu repräsentieren (Bishop 2006). Künstliche neuronale Netze (ANN) versuchen die Funktionsweise von biologischen Systemen, beispielsweise des menschlichen Gehirns, nachzubilden. Solche künstlichen neuronalen Netze können laut (Yin 2008) nicht nur bessere Ergebnisse als traditionelle Problem-Löse-Verfahren liefern, sondern geben uns auch ein besseres Verständnis der geistigen Fähigkeiten des menschlichen Gehirns. Laut (Yegnanarayana 2009, 15) ist es allerdings noch ein weiter Weg, bis ANNs ein biologisches neuronales Netz gänzlich nachbilden können. Hauptgrund dafür ist die enorme Anzahl von Neuronen und Synapsen, sowie deren asynchrone Interaktion, die simuliert werden müsste. Doch auch mit der heutigen Rechenleistung ist es durchaus möglich, Probleme mit Hilfe von ANNs zu lösen.

3.5.1 Biologischer Hintergrund

Der im menschlichen Gehirn am höchsten entwickelte Teil, von dem auch unsere Intelligenz ausgeht, ist der Neocortex, ein Teil der Großhirnrinde. Dieser hat eine ungefähre Fläche von 0,2 Quadratmetern, ist circa 2-3 Millimeter dick und stark verworren, um Platz zu sparen. Im Neocortex können verschiedene Bereiche unterschieden werden, die auf bestimmte Aufgaben spezialisiert sind, beispielsweise visuelle Wahrnehmung oder Bewegung (Ritter u. a. 1992, 12). Laut (Andersen, Korbo und Pakkenberg 1992) verfügt die menschliche Großhirnrinde über 21 bis 26 Milliarden Neuronen.

Abbildung 12 zeigt die Entwicklung des *SyNAPSE* Chips, der von IBM entwickelt wird. Die zweite Generation des Chips, die im August 2014 vorgestellt wurde, verfügt bereits über 1 Million programmierbarer Neuronen und kann 256 Millionen Synapsen Hardwareunterstützt simulieren. Um damit die menschliche Großhirnrinde, und damit deren Intelligenz zu simulieren, wäre ein Verbund von rund 26000 dieser Chips nötig. Die Entwicklung innerhalb von nur drei Jahren von der ersten auf die zweite Generation zeigt jedoch, dass dieses Ziel in greifbare Nähe rückt.

Abbildung 12: IBM SyNAPSE Chip
(Quelle: http://www.research.ibm.com/cognitive-computing/neurosynaptic-chips.shtml)

3.5.2 Anwendung

Das Ziel von ANNs ist es, die Art, wie natürliche neuronale Netze Wahrnehmungs- oder Wiedererkennungs-Aufgaben lösen, nachzubilden. Damit ein neuronales Netz eine bestimmte Aufgabe lösen kann, müssen die Neuronen durch eine enorme Anzahl von gewichteten Synapsen verbunden werden. Das Verbinden der Neuronen und die Gewichtung der entstehenden Synapsen erfolgt durch ein Lern-Verfahren beziehungsweise einen Trainings-Algorithmus (Yin 2008).

Ein trainiertes ANN soll laut (Zupan, Novič und Ruisánchez 1997) in der Lage sein, eine der folgenden Aufgaben beziehungsweise Probleme zu lösen:

1. Für jedes gegebene Signal $X_s = (x_{s1}, x_{s2}, ..., x_{sm})$ dasjenige Ausgabe-Signal $Y_s = (y_{s1}, y_{s2}, ..., y_{sn})$ zu finden, das dem vordefinierten Ziel-Vektor $T_s = (t_{s1}, t_{s2}, ..., t_{sn})$ am ähnlichsten ist.

2. Für jedes gegebene Signal $X_s = (x_{s1}, x_{s2}, ..., x_{sm})$ mit der Klasse q_s, alle Neuronen, die dieser Klasse entsprechen, zu aktivieren.

3. Ohne das vorherige Wissen über die intrinsischen Eigenschaften der Eingabe-Objekte, eine zweidimensionale topologische Distribution (2D Karte), mit den am meisten aktiven Ausgabe-Signalen (Neuronen), als Ergebnis für alle r Eingabe-Objekte $X_s = (x_{s1}, x_{s2}, ..., x_{sm}), s = 1, ..., r$, zu erzeugen.

Um die ersten beiden Aufgaben zu lösen, wird überwachtes Lernen eingesetzt (siehe 3.4). Dafür wird ein Set von Eingabe-Ausgabe-Paaren $\{X_s, T_s\}$, wobei $X_s = (x_{s1}, x_{s2}, ..., x_{sm})$ einem m-dimensionalen Eingabe-Objekt und $T_s = (t_{s1}, t_{s2}, ..., t_{sn})$ dem dazugehörigen Ziel-Vektor entspricht, benötigt. Während des überwachten Lernens wird für jedes Eingabe-Objekt X_s der Ausgabe-Vektor Y_s berechnet und mit dem Ziel-Vektor T_s verglichen. Nach diesem Vergleich werden, entsprechend der jeweiligen ANN-Strategie, die Gewichte der Neuronen so angepasst, dass im finalen Modell Y_s und T_s bestmöglich übereinstimmen.

Die letzte Aufgabe (2D Karte) wird durch nicht überwachtes Lernen (siehe 3.4) gelöst, wobei keine Ziel-Vektoren T_s im Voraus bekannt sind. Diese können allerdings, falls vorhanden, genutzt werden, um das gelernte Modell im Nachhinein zu überprüfen.

In jedem Fall ist es nötig, alle Eingabe-Objekte X_s des Trainings-Sets (und Ziel-Vektoren T_s wenn benötigt) mehrfach auf das Netzwerk anzuwenden, um dieses zu trainieren. Das Anpassen aller Gewichtungen im ANN durch alle Eingabe-Objekte entspricht dabei einem Zyklus, auch Epoche genannt, des Lern-Prozesses. Das Anwenden aller Eingabe-Objekte wird so lange wiederholt, bis entweder die Distanz zwischen allen T_s und Y_s akzeptabel ist oder eine definierte Anzahl von Epochen erreicht wurde.

3.6 Selbst-organisierende Karten

Unter den existierenden ANN Architekturen und Algorithmen, ist die Self-Organizing Map (SOM) oder auch *Kohonen ANN* von (Kohonen 1990) eines der populärsten Modelle. Dabei handelt es sich um ein abstraktes mathematisches Modell zur topographischen Zuordnung von (visuellen) Reizen durch ein neuronales Netzwerk (Yin 2008).

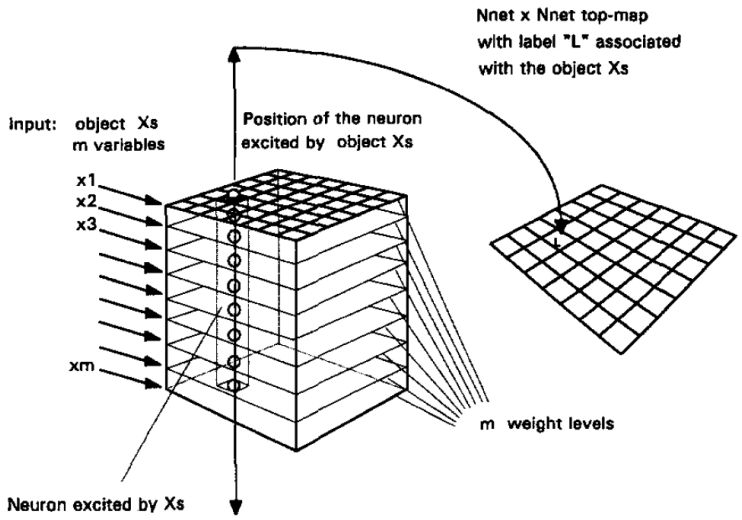


Abbildung 13: Selbst-Organisierende Karte (Zupan, Novič und Ruisánchez 1997)

Abbildung 13 zeigt eine SOM oder Kohonen-Karte. Die Neuronen werden dabei als Säulen dargestellt und sind in einer quadratischen Matrix ($N_{net} \times N_{net}$) angeordnet. Die Länge der Säulen (m) entspricht dabei der Anzahl von Gewichten beziehungsweise der Anzahl der Dimensionen der Eingabe-Vektoren X_s. Alle Gewichte der i-ten Ebene erhalten gleichzeitig die i-te Komponente x_i des Eingabe-Vektors X_s. Das erregte Neuron c mit dem Label „L“ wird aufgrund der besten Übereinstimmung gewählt.

Das Kohonen ANN basiert auf einer einzigen Schicht von Neuronen, die quadratisch angeordnet sind und einer zweidimensionalen Fläche, die Reaktionen auf der Oberseite anzeigt (siehe Abbildung 13). Jedes Neuron in diesem Schema besitzt 8, 16, 24, usw. Nachbarn in seiner ersten, zweiten, dritten, usw. Ebene (Zupan, Novič und Ruisánchez 1997).

3.6.1 Training

Während des Lernvorganges wird derselbe multidimensionale Vektor X_s allen Neuronen als Eingabe übergeben. Für jedes Neuron wird dieser Vektor X_s mit dessen Gewichten verglichen. Die Gewichte wurden dafür vor dem Lernprozess zufällig mit $0 <= w <= 1$ initialisiert. Für Kohonen ANNs gilt die Regel, dass ein Eingabe-Objekt nur ein Neuron stimulieren darf. Aus diesem Grund konkurrieren die Neuronen untereinander darum, welches schlussendlich erregt wird. Es wird daher auch von *competitive learning* gesprochen. Das Neuron, das aus der $N_{net} \times N_{net}$ Matrix als Sieger hervorgeht, wird als *winning* oder *central* Neuron c bezeichnet. Die Auswahl des gewinnenden Neurons basiert auf dem Vergleich der Gewichts-Vektoren $W_j = (w_{j1}, w_{j2}, ..., w_{jm})$ aller Neuronen mit dem Eingabe-Signal $X_s = (x_{s1}, x_{s2}, ..., x_{sm})$ und ist folgendermaßen definiert:

$$Neuron\ c \leftarrow \min \left\{ \sum_{i=1}^{m} (x_{si} - w_{ji})^2 \right\}, \quad j = 1, 2, ..., c, ..., N_{net} \times N_{net} \tag{1}$$

Nachdem das gewinnende Neuron c gefunden wurde, startet die Korrektur der Gewichte (lernen). Das charakteristische Merkmahl des Kohonen ANN ist die Implementierung dieser Korrektur, welche einem natürlichen neuronalen Netz stark nachempfunden wurde. Die Korrektur der Gewichte betrifft hierbei nicht das gesamte Netzwerk und nicht einmal die selbe Anzahl von Neuronen in verschiedenen Phasen des Lernprozesses. Es werden nur die Gewichte der Neuronen aktualisiert, die sich in topologischer Nähe zum gewinnenden Neuron befinden. Dieses *local feed-back* Verfahren bringt alle topologisch nahen Neuronen dazu, sich ähnlich zu verhalten, wenn dem ANN ähnliche Eingabe-Objekte übergeben werden. Das bedeutet, dass durch zwei ähnliche Objekte auch zwei topologisch nahe Neuronen erregt werden. Damit das ANN lernt, müssen die Gewichte W_{ci} des Neurons c in jeder Epoche so angepasst werden, dass diese in den folgenden Epochen die Eingabe-Objekte X_s immer besser widerspiegeln (siehe Gleichung 1). Wie bereits erwähnt, werden nicht nur das gewinnende Neuron c, sondern auch dessen $8, 16, 24, ..., 8p$ Nachbarn in dessen ersten, zweiten, dritten, ... p-ten Nachbarschaft stimuliert. Jedoch hängt es von den Parametern der Lern-Strategie a_{max}, a_{min}, N_{net} und i_{tot} ab, welche benachbarten Neuronen und in welchem Umfang diese stimuliert werden:

$$\begin{aligned} \Delta w_{ji} = &[(a_{max} - a_{min})(p/N_{net}) + a_{min}] \\ &\times [1 - d/(p+1)] \left(x_{si} - w_{ji}^{old}\right), \quad d = 0, 1, 2, ..., p \end{aligned} \tag{2}$$

Gleichung 2 beschreibt, wie die Korrektur des Gewichtes w_{ji} mit steigender Lernzeit und der steigenden topologischen Distanz d zwischen dem j-ten und dem gewinnenden Neuron c, abnimmt. Das Intervall p, das bestimmt, wie viele Neuronen in der Nachbarschaft stimuliert werden, nimmt dafür während des Lernvorganges folgendermaßen ab:

$$p = (i_{tot} - i_{it}) N_{net} / (i_{tot} - 1) \tag{3}$$

Am Beginn des Lernvorganges ($i_{it} = 1$) deckt p das gesamte Netzwerk ($p = N_{net}$) ab. Am Ende ($i_{it} = i_{tot}$) beschränkt sich p nur noch auf das gewinnende Neuron c ($p = 0$). Egal ob die Differenz zwischen $\left(x_{si} - w_{ji}^{old}\right)$ in Gleichung 2 positiv oder negativ ist, wird w_{ji}^{new} nach jeder Epoche näher an x_{si} sein als an w_{ji}^{old}. Das nicht überwachte Lernen wird gewöhnlich für eine vorbestimmte Anzahl an Iterationen oder Epochen (i_{tot}) durchgeführt (Zupan, Novič und Ruisánchez 1997).

Top-Map

Nachdem das Kohonen ANN alle Epochen (i_{tot}) der Trainingsphase durchlaufen hat, werden die gesamten Eingabe-Objekte ein weiteres Mal dem ANN übergeben. In diesem letzten Durchlauf werden die Labels der Eingabe-Objekte in der *Top-Map* den jeweils erregten Neuronen zugewiesen. Diese *Top-Map* kann als einfache, zweidimensionale Visualisierung der hochdimensionalen Daten betrachtet werden. (Zupan, Novič und Ruisánchez 1997).

3.7 Counterpropagation Artificial Neural Networks

Was den Aufbau und die Lern-Strategie angeht, sind CP-ANN und Kohonen ANN sehr ähnlich. Ein CP-ANN besteht aus zwei Schichten, einer Kohonen- und einer Ausgabe-Schicht. Beide Schichten haben exakt die selbe Anzahl und Anordnung von Neuronen, welche auch exakt übereinander angeordnet sind. Jedes Neuron der Kohonen-Schicht hat somit eine eins-zu-eins Beziehung zum jeweiligen Neuron der Ausgabe-Schicht (siehe Abbildung 14). Das CP-ANN stellt somit eine Erweiterung des Kohonen ANN beziehungsweise SOM dar. Der Zweck dieser Erweiterung ist es, mit Hilfe eines Kohonen ANN auch Probleme des überwachten Lernens (siehe 3.4) zu lösen. Auch wenn die Ausgabe-Schicht über exakt die selbe Anzahl und Anordnung von Neuronen wie die Kohonen-Schicht verfügt, unterscheidet sich die Anzahl der Gewichte dieser Neuronen. In der Ausgabe-Schicht entspricht die Anzahl der Gewichte von Neuronen den Dimensionen der Ziel-Vektoren T_s (Zupan, Novič und Ruisánchez 1997).

Abbildung 14 zeigt ein CP-ANN, welches aus zwei Neuronen-Schichten, der Kohonen- und der Ausgabe-Schicht, besteht. Das gewinnende Neuron c wird für Eingabe-Objekte X_s in der Kohonen-Schicht ermittelt. Die Gewichtungen der benachbarten Neuronen werden anschließend in der Kohonen *und* der Ausgabe-Schicht angepasst.

3.7.1 Training

In CP-ANN werden die Gewichte der Neuronen der Kohonen- und Ausgabe-Schicht laut (Zupan, Novič und Ruisánchez 1997) mittels der sehr ähnlichen Gleichungen 4 und 5 korrigiert:

$$\begin{aligned} \Delta w_{ji}^{Kohonen} = &[(a_{max} - a_{min})(p/N_{net}) + a_{min}] \\ &\times [1 - d/(p+1)] \left(x_{si} - w_{ji}^{Kohonen}\right), \quad d = 0, 1, 2, ..., p \end{aligned} \tag{4}$$

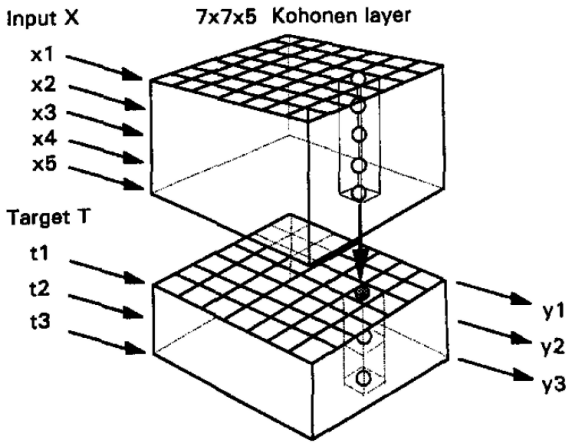


Abbildung 14: Counterpropagation Artificial Neural Network (Zupan, Novič und Ruisánchez 1997)

$$\begin{aligned}\Delta w_{ji}^{output} = &[(a_{max} - a_{min})(p/N_{net}) + a_{min}] \\ &\times [1 - d/(p+1)] \left(t_{si} - w_{ji}^{output}\right), \quad d = 0, 1, 2, ..., p\end{aligned} \tag{5}$$

Beide Gleichungen wurden von Gleichung 2, die zum Korrigieren der Gewichte im Kohonen ANN verwendet wird, abgeleitet. Die Gleichungen 4 und 5 werden auf die Gewichte $w_{ji}^{Kohonen}$ und w_{ji}^{output} der Eingabe- (Kohonen) und Ausgabe-Schicht des CP-ANN Modells angewandt. Gleichung 2 und 4 sind dabei identisch, während Gleichung 5 leicht modifiziert wurde, um überwachtes Lernen und damit Ziel-Vektoren T_s anstelle von Eingabe-Objekten X_s zu unterstützen. Einen entscheidenden Unterschied stellt hierbei die Wahl des gewinnenden Neurons c und damit auch die Wahl der betroffenen benachbarten Neuronen dar. Bei Gleichung 4 wird c genau wie beim Kohonen ANN gewählt. Mittels der Gleichung 1 wird das Neuron gesucht, dessen Gewichte dem Eingabe-Vektor $X_s = (x_{s1}, x_{s2}, ..., x_{sm})$ am ähnlichsten sind. Wenn das gewinnende Neuron c in der Eingabe-Schicht (Kohonen) gefunden wurde, gilt dieses auch für die Ausgabe-Schicht. Das bedeutet, dass die topologische Position des Neurons c in beiden Schichten gleich ist. Die Gewichte in der Ausgabe-Schicht werden anschließend mittels der Komponenten t_{si} des Ziel-Vektors $T_s = (t_{s1}, t_{s2}, ..., t_{sn})$ anstelle von x_{si} der Eingabe-Objekte, korrigiert (Zupan, Novič und Ruisánchez 1997).

Nachdem das Training beendet wurde, kann mit Hilfe der Kohonen-Schicht im CP-ANN für jedes Eingabe-Objekt X_s die Position des Neurons in der Ausgabe-Schicht ermittelt werden. Aufgrund der Tatsache, dass auch die Ziel-Klassen während des Lern-Prozesses *verteilt* werden (siehe Gleichung 5), enthalten alle Neuronen der Ausgabe-Schicht Klassen, auch wenn das dazugehörige Neuron der Kohonen-Schicht niemals während des Trainings erregt wurde (Zupan, Novič und Ruisánchez 1997).

4 AndroSOM Feature Miner

In dieser Arbeit werden statische sowie dynamische Analyse eingesetzt, um ein möglichst gutes Gesamtbild der zu analysierenden Applikationen zu erhalten. Dazu wurde ein Framework namens „AndroSOM Feature Miner“ entwickelt. Das Grundgerüst wurde in der Programmiersprache *GO*[13] implementiert und verfügt über ein GUI. Mit Hilfe dieses GUI können alle nötigen Schritte durchgeführt werden, um verschiedene Datensätze zu erstellen, welche später in *MatLab* analysiert werden können. Das Framework zum Extrahieren von Features wurde in drei Module eingeteilt:

1. **Modul zum Sammeln von Metadaten:** Metadaten werden von *VirusTotal*[14] bezogen. Mit Hilfe dieser Daten wird entschieden, ob es sich bei einem bestimmten Sample um Schadsoftware handelt oder nicht.

2. **Modul zur statischen Analyse:** Die statische Analyse wird mit Hilfe von *Androguard* durchgeführt, einem weit verbreiteten Framework für die statische Analyse von Android Applikationen (siehe 3.2.1 und 3.2.3). Daten aus der statischen Analyse fließen nicht nur direkt in die später erzeugten Feature-Vektoren ein, sondern werden auch für die dynamische Analyse verwendet, um beispielsweise alle Dienste eines Samples zu starten oder bestimmte Broadcasts auszusenden, auf die eine Applikation hört.

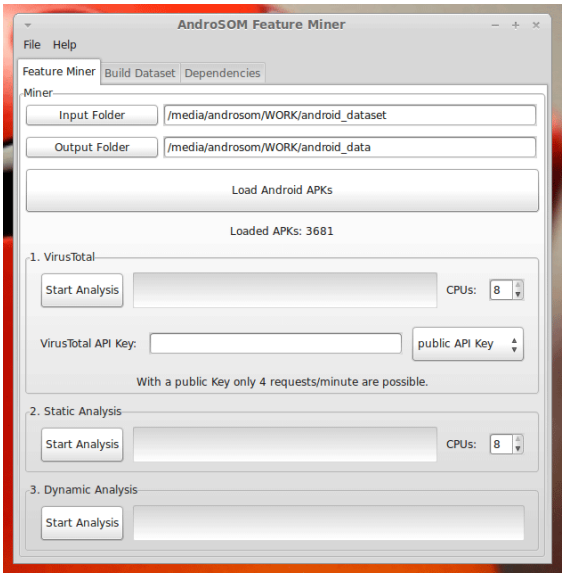
3. **Modul zur dynamischen Analyse:** Besonderes Augenmerk lag bei der dynamischen Analyse darauf, diese möglichst performant zu gestalten. Deshalb wird nicht, wie in vielen anderen Arbeiten (siehe 3.2.2 und 3.2.3), *QEMU* als Emulator verwendet, sondern auf *VirtualBox* und *Android-x86* gesetzt. Während der dynamischen Analyse wird nicht nur das Verhalten eines Samples während dessen Ausführung überwacht, sondern auch ein möglicher Netzwerk-Verkehr analysiert.

Die Module zum Sammeln von Metadaten und zur statischen sowie dynamischen Analyse wurden größtenteils in *Python* beziehungsweise in Form von *Shell*-Skripten implementiert.

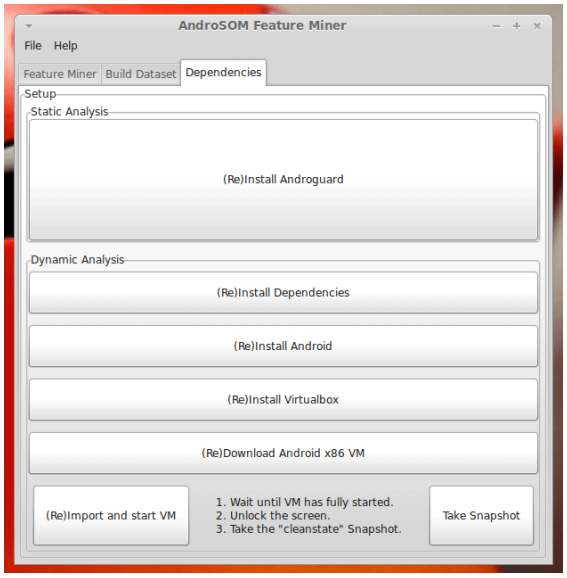
4.1 AndroSOM GUI

Das GUI wurde mit Hilfe von *GTK*[15] implementiert und ist in drei Bereiche aufgeteilt, die über *Tabs* (siehe Abbildung 15) zu erreichen sind. Um das AndroSOM GUI zu starten, kann im Stammverzeichnis des Frameworks `./featureminer` ausgeführt werden. Dabei handelt es sich um eine für die Linux x86-Architektur kompilierte Binärdatei. Ansonsten kann das AndroSOM Framework auch mittels des Befehls `go run featureminer` im Stammverzeichnis des Frameworks aufgerufen werden. Dafür ist allerdings eine vorangehende Installation der *GO* Entwicklungsumgebung von Nöten.

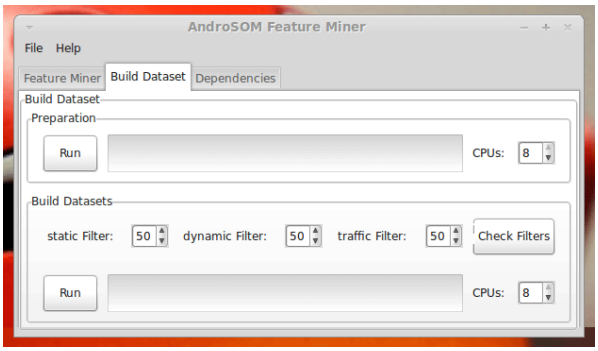
13. https://golang.org
14. https://www.virustotal.com
15. http://www.gtk.org



(a) Miner Tab



(b) Dependencies Tab



(c) Dataset Tab

Abbildung 15: AndroSOM: GUI

Installation der Abhängigkeiten

Wie in Abbildung 15b zu sehen ist, können über das AndroSOM GUI alle benötigten Abhängigkeiten installiert, sowie das AndroSOM Android 4.4 x86 System-Image heruntergeladen und in *VirtualBox* importiert werden. Abschließend wird diese AndroSOM Sandbox für die weitere Verwendung eingerichtet. Dazu müssen die AndroSOM Sandbox gebootet und der *Lockscreen* entsperrt werden. Durch den anschließenden Klick auf „Take Snapshot“ wird ein Abbild des momentanen, sauberen System-Zustandes angefertigt und die VM automatisch heruntergefahren. Dieses Abbild wird in weiterer Folge vor jedem dynamischen Analyse-Durchgang wiederhergestellt, damit jedes Sample auf einem sauberen und unveränderten System ausgeführt wird.

Die Installation aller notwendigen Komponenten beziehungsweise Abhängigkeiten durch vorgefertigte Setup-Scripte und die Einrichtung der AndroSOM Sandbox für die dynamische Analyse wurde unter „Linux Mint 16 64-bit“ getestet. Sollte es zu Problemen

kommen, kann die genaue Auflistung aller benötigten Pakete und deren Installations-Verzeichnis den jeweiligen Setup-Scripten unter `./scripts/setup/` entnommen werden.

Extrahieren der Features

Um Features zu extrahieren, kann nach der Installation der Abhängigkeiten und der Einrichtung der AndroSOM Sandbox der „Feature Miner“ Tab (siehe Abbildung 15a) verwendet werden. Dieser gliedert sich in vier Bereiche: Einen allgemeinen Bereich, in dem das Input- und ein Output-Verzeichnis zu wählen ist und drei Bereiche zum Steuern der zuvor genannten Module.

Das Input-Verzeichnis enthält alle zu analysierenden Samples in Form von APK-Dateien. Diese können sich auch in Unterordnern befinden. Im Output-Verzeichnis werden, beispielsweise während der dynamischen Analyse, alle gesammelten Log-Dateien in Unterverzeichnissen für jedes Sample gesichert, damit diese gegebenenfalls auch für eine spätere Analyse durch externe Anwendungen zur Verfügung stehen. Das Sichern der *Rohdaten* erschien sinnvoll, da ein erneuter Durchlauf der dynamischen Analyse sehr zeitaufwendig wäre.

Nach dem Auswählen der Verzeichnisse können die Samples durch einen Klick auf „Load Android APKs“ geladen werden. Ab diesem Zeitpunkt stehen alle drei Module zur Verfügung. Diese müssen jedoch in der gegebenen Reihenfolge ausgeführt werden, da sie aufeinander aufbauen.

- **1. VirusTotal:** Dieses Modul dient zum Sammeln von Metadaten. Wird ein gültiger API-Schlüssel für die private API angegeben, können die benötigten Daten sehr schnell parallel abgefragt werden. Verfügt der oder die BenutzerIn nur über einen öffentlichen API-Schlüssel, sind nur 4 Anfragen pro Minute möglich. Auch das Einreichen von Samples, die über 32 Megabyte groß sind, funktioniert nur über die private API. Ein öffentlicher API-Schlüssel wird allen auf *VirusTotal*[16] angemeldeten BenutzerInnen gratis zur Verfügung gestellt.
- **2. Static Analysis:** Dieses Modul führt eine statische Analyse einer APK-Datei mit Hilfe von *Androguard* durch. Es können maximal so viele Samples parallel analysiert werden, wie CPU-Kerne auf dem System zur Verfügung stehen.
- **3. Dynamic Analysis:** Dieses Modul führt eine dynamische Analyse durch. Auf die Funktionsweise der dynamischen Analyse wird in Kapitel 4.2.3 genau eingegangen.

Datensätze erstellen

Um Datensätze für die spätere Analyse in *MatLab*[17] zu erstellen, verfügt der „Feature Miner“ über einen Tab namens „Build Dataset“ (siehe Abbildung 15c).

16. https://www.virustotal.com
17. http://www.mathworks.de/products/matlab/

Hier stehen wiederum zwei Module zur Verfügung, welche in der gegebenen Reihenfolge ausgeführt werden müssen. Um diese Module zu benutzen, ist es nötig, dass im „Feature Miner“ Tab ein Input- und Output-Verzeichnis gewählt und die Samples geladen wurden, die später in den Datensätzen enthalten sein sollen. So können auch kleinere Datensätze, die nicht alle analysierten Samples enthalten, erstellt werden.

1. **Preparation:** Im ersten Schritt werden die gesamten Features aller geladenen Samples in einer zentralen Datenbank gesammelt, zusammen mit der Häufigkeit ihres Vorkommens in schädlichen und legitimen Applikationen und anderen Metadaten.

2. **Build Datasets:** Im zweiten Schritt werden mit Hilfe der zuvor generierten Feature-Datenbank spezifische Feature-Vektoren für alle geladenen Samples erstellt. Diese Feature-Vektoren werden anschließend in verschiedene Datensätze geschrieben. Dabei kann für alle drei Arten von Features (aus der statischen und dynamischen Analyse, wobei bei der dynamischen Analyse die Daten aus dem Netzwerk-Verkehr extra behandelt werden) ein Filter gesetzt werden, welcher angibt, in wie vielen Samples ein bestimmtes Feature vorhanden sein muss, um in die entsprechenden Datensätze aufgenommen zu werden. Mit einem Klick auf „Check Filters“ kann überprüft werden, wie viele Features sich in den resultierenden Datensätzen, bei den momentan gesetzten Filter-Werten, befinden.

4.2 Architektur des Frameworks

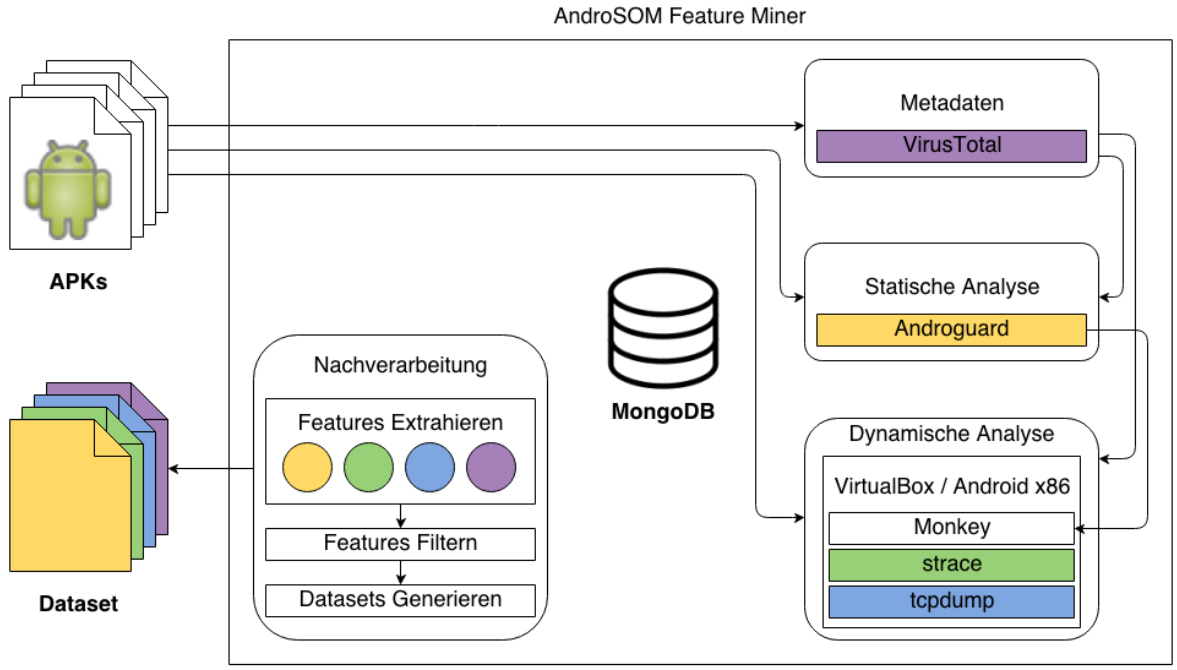


Abbildung 16: AndroSOM: Architektur

Wie bereits zuvor beschrieben, besteht das Framework aus einem Grundgerüst und verschiedenen Modulen, die ineinander greifen. Für die Implementierung des Frameworks wurden dabei unterschiedliche Programmiersprachen verwendet. Das Grundgerüst wurde dabei in *GO* geschrieben, während es sich bei den Modulen größtenteils um *Python*- beziehungsweise *Shell*-Skripte handelt. Damit wurde versucht, für jede Aufgabe das richtige Werkzeug zu wählen.

Abbildung 16 zeigt, wie der „AndroSOM Feature Miner“ grob aufgebaut ist. Im Zentrum steht eine NoSQL Datenbank (MongoDB), in die alle Module ihre generierten Daten als Sammlungen von JavaScript Object Notation (JSON)-ähnlichen Dokumenten speichern. Diese Daten können in weiterer Folge bei Bedarf effizient von den anderen Modulen abgefragt werden.

Zuerst müssen Metadaten zu den Samples gesammelt werden. Diese stammen von *VirusTotal* und werden in der Datenbank abgelegt, sofern diese noch nicht vorhanden sind. Jedes Sample wird dafür eindeutig durch dessen Secure Hash Algorithm - Version 1 (SHA-1) Hash identifiziert. Die so gesammelten Daten von *VirusTotal* finden hauptsächlich für die Filterung von Samples Verwendung, bei denen es sich nicht eindeutig genug um Schadsoftware (von mindestens 35 Viren-Scannern erkannt) oder legitime Applikationen (von keinem einzigen Viren-Scanner erkannt) handelt. 35 positive Treffer, bei momentan 50 Viren-Scannern, erschien geeignet, da diese Zahl relativ hoch ist, aber trotzdem nicht allzu viele Samples aus der Wertung fallen (siehe auch Abbildung 24). Alle anderen Module greifen auf diese Daten zu, um nicht brauchbare Samples zu überspringen. Die statische Analyse benötigt als weiteren Input nur eine APK-Datei und legt die mittels *Androguard* gesammelten und aufbereiteten Daten anschließend in eine MongoDB Collection (Sammlung von Dokumenten) ab. Die so generierten Daten werden unter anderem von der dynamischen Analyse verwendet, um bestimmte Events, wie beispielsweise Broadcasts zu simulieren oder einfach alle in einem Sample enthaltenen Dienste gleich am Beginn der Ausführungszeit zu starten. Zusätzlich werden zufällige Events durch den *Android Monkey* (siehe Kapitel 3.3) generiert. Die bei der dynamischen Analyse mittels `strace` und `tcpdump` gesammelten Daten werden aufbereitet und in separaten MongoDB Collections abgelegt.

Nach dem Analysieren der Samples wird eine *Referenz*-Collection angelegt, die alle Features zusammen mit Metadaten enthält (siehe Listing 7). Bei diesen Metadaten handelt es sich um die Anzahl der Vorkommen eines Features in legitimen Applikationen beziehungsweise in Schadsoftware. Wenn nötig, wird auch ein Maximal-Wert, der beschreibt, wie oft ein Feature in einem Sample maximal vorkommt, abgespeichert. Dieser kann für die spätere Normalisierung der Daten benötigt werden. Die gesammelten Metadaten werden außerdem für die Filterung von Features verwendet. Als letzter Schritt werden verschiedene Datensätze generiert, die für die weitere Analyse in *MatLab* importiert werden können.

Das Grundgerüst wurde in *GO* geschrieben und besteht unter anderem aus einem GUI (siehe Abbildung 15), sowie mehreren *GO-Packages*, beispielsweise (`./miner/miner.go`), mit dessen Hilfe *Python*-Scripte effizient parallel ausgeführt und das GUI aktualisiert

werden können. Hierfür stellt *GO* sogenannte *Channels*[18] zur Verfügung, womit ein Producer/Consumer Pattern, wie es in Listing 2 zu sehen ist, implementiert wurde.

Listing 2: Producer/Consumer Pattern in GO

```
type Job struct {
    command string
    args    []string
}

func Producer(apks *[]string, outputFolder string,
              script string, numCPU int) {
    numOfJobs := len(*apks)
    jobsChan := make(chan Job, numOfJobs)
    doneChan := make(chan int)

    go func() {
        for _, path := range *apks {
            args := []string{
                "-i" + path,
                "-o" + outputFolder,
            }
            job := Job{command: working_dir + "/scripts/" + script,
               args: args}
            jobsChan <- job
        }
    }()

    for i := 0; i < numCPU; i++ {
        go consumer(jobsChan, doneChan)
    }

    updateGui(doneChan, progressBar, numOfJobs)
}

func consumer(jobsChan chan Job, doneChan chan int) {
    for {
        job := <-jobsChan
        cmd := exec.Command(job.command, job.args...)
        if err := cmd.Run(); err != nil {
            fmt.Println(err)
        }
        doneChan <- True
    }
}
```

18. https://golang.org/doc/effective_go.html

Die `Producer` Methode wird dabei nur einmal durch das GUI aufgerufen. Ihr werden unter anderem ein String-Array mit den Pfaden aller geladenen APK-Dateien, der Pfad zum jeweiligen Analyse-Script (beispielsweise statische oder dynamische Analyse) und die Anzahl zu startender `Consumer` übergeben. Zuerst werden zwei *Channels* angelegt. Durch den `jobsChan` werden nacheinander die Pfade von allen geladenen APK Dateien geschickt, während durch den `doneChan` immer dann ein `True` gesendet wird, wenn das auszuführende *Python*-Script beendet wurde. Um den `jobsChan` zu befüllen, wird eine *Go-Routine*, welche in einem extra Thread asynchron ausgeführt wird, gestartet. Parallel dazu wird eine beliebige Anzahl von `Consomer`-Methoden, ebenfalls als *GO-Routinen*, gestartet. Der `Producer` wartet anschließend, bis alle `Job`-Instanzen abgearbeitet wurden und aktualisiert gegebenenfalls das GUI. Die `Consumer` nehmen nacheinander alle `Job`-Instanzen entgegen, arbeiten diese ab und benachrichtigen den `Producer`, wenn ein Job fertiggestellt wurde.

Neben dem GUI und dem *Miner*-Package ist die in `./tools/mongodb/` installierte *MongoDB* Datenbank einer der Grundpfeiler des Frameworks. Diese läuft auf einem eigens konfigurierten Port und legt ihre Daten in `./data/` ab, wodurch es zu keinen Konflikten kommen sollte, auch wenn *MongoDB* bereits auf dem System installiert sein sollte. Die Datenbank wird automatisch mit dem GUI gestartet beziehungsweise beendet.

Weitere Grundpfeiler sind das unter `./tools/androguard/` installierte *Androguard* Framework, welches für die statische Analyse verwendet wird, sowie eine für den lokalen Benutzer installierte *VirtualBox*- und Android Entwicklungs-Umgebung. All diese Abhängigkeiten können über die entsprechenden Installations-Scripte in `./scripts/setup/` oder direkt über das GUI (siehe Abbildung 15b) installiert werden.

4.2.1 Modul: Metadaten

Ob es sich bei einem Sample um Schadsoftware handelt oder nicht, wird mit Hilfe von *VirusTotal* entschieden. Von den in dieser Arbeit verwendeten Samples befanden sich bereits rund 95% in der *VirusTotal* Datenbank und mussten somit nicht mehr extra analysiert werden. Die Daten standen daher in den meisten Fällen ohne weitere Wartezeit zur Verfügung.

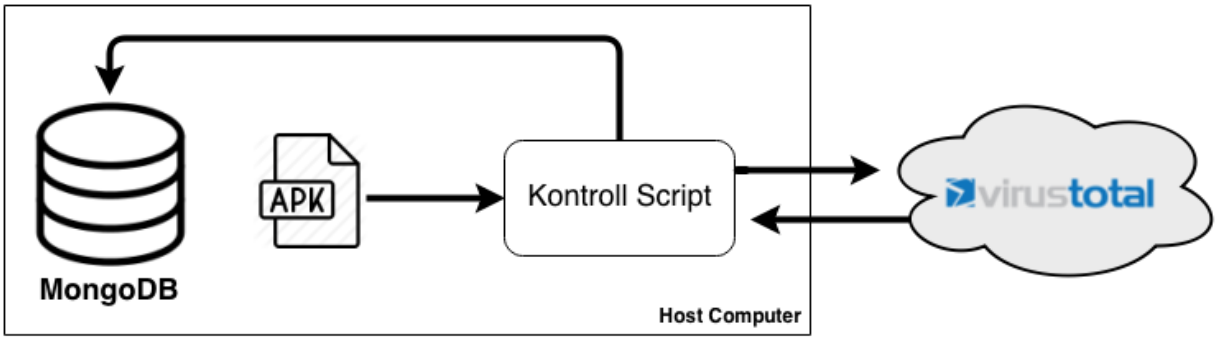


Abbildung 17: AndroSOM: Sammeln von Metadaten

In Abbildung 17 wird die Funktionsweise des Metadaten-Moduls vereinfacht dargestellt. Dem Kontrollscript wird eine APK-Datei übergeben. Mit dieser wird ein SHA-1 Hash generiert, der genutzt wird, um einen Bericht für das entsprechende Sample von *VirusTotal* anzufordern. Wird von der *VirusTotal*-API der Response-Code `0` zurückgegeben, handelt es sich um ein unbekanntes Sample, welches zur weiteren Analyse durch eine Vielzahl von Viren-Scannern hochgeladen werden muss. Kommt als Response-Code `1` zurück, können die erhaltenen Daten (siehe Listing 3) direkt in die entsprechende *MongoDB* Collection geschrieben werden. Der Response-Code `-2` bedeutet, dass sich das bereits hochgeladene Sample in der Warteschlange befindet und die Daten zu einem späteren Zeitpunkt bezogen werden können.

Wenn ein Sample zur Analyse an die *VirusTotal*-API geschickt werden muss, sollte es kleiner als 32MB sein. Für größere Samples muss über die private API eine *Upload-URL* angefordert werden, wofür ein privater API-Schlüssel von Nöten ist. Da nur Metadaten für Samples angefordert werden, die sich noch nicht in der Datenbank befinden, kann dieses Modul so oft hintereinander ausgeführt werden, bis für jedes Sample auf *VirusTotal* ein Bericht vorliegt.

Gesammelte Daten

Die Metadaten werden direkt von der *VirusTotal*-API bezogen und unverändert in einer *MongoDB* Collection abgelegt. Listing 3 zeigt einen Auszug der Daten, welche von *VirusTotal* durch Angabe eines Hashes bezogen werden können. Besonders wichtig hierbei sind die `positives`. Dieser Wert gibt an, von wie vielen Viren-Scannern eine Datei als schädlich klassifiziert wurde. Im Zuge dieser Arbeit wurde ein Grenzwert von mindestens 35 Viren-Scannern festgelegt, die ein Sample als Schadsoftware klassifiziert haben müssen, beziehungsweise 0 positive Treffer für legitime Applikationen (siehe auch 24). Samples, die von zwischen 1 bis 34 Viren-Scannern positiv klassifiziert wurden, werden nicht berücksichtigt.

Listing 3: VirusTotal API Response

```
{
  "sha1": "fc16aba5d573312489911294aaf7cf9b439bbdc9",
  "response_code": NumberInt(1),
  "positives": NumberInt(40),
  "total": NumberInt(53),
  "scan_date": "2014-08-19 11:53:10",
  "scans": {
    "Bkav": {
      "detected": true,
      "result": "MW.Clod81b.Trojan.bc0e",
    },
    ...
  }
}
```

4.2.2 Modul: Statische Analyse

Für die statische Analyse wird *Androguard* eingesetzt. Wie in Abbildung 18 zu sehen ist, wird auch dem für die statische Analyse zuständigen *Python*-Script eine APK-Datei übergeben. Die Analyse wird gleich zu Beginn abgebrochen, wenn für das übergebene Sample keine *VirusTotal* Metadaten gefunden werden können oder es sich nicht eindeutig genug um Schadsoftware beziehungsweise eine legitime Applikation handelt.

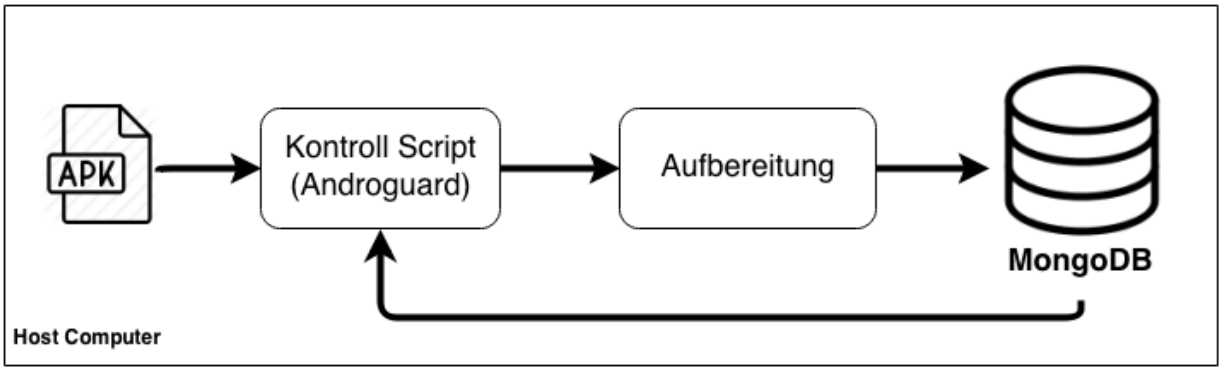


Abbildung 18: AndroSOM: Statische Analyse

Wenn die Analyse nicht zuvor abgebrochen wurde, wird das Sample entpackt und anschließend durch *Androguard* festgestellt, ob es sich dabei um eine valide APK-Datei handelt. Wenn ja, wird die statische Analyse durchgeführt. Anschließend werden die Ergebnisse aufbereitet und in einer *MongoDB* Collection geschrieben. Die statische Analyse kann, wie auch das vorangegangene Sammeln von Metadaten, parallelisiert werden.

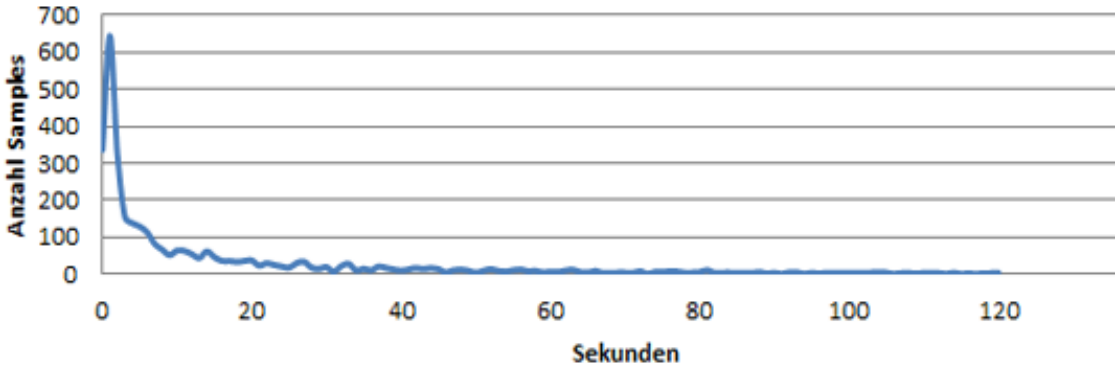


Abbildung 19: Laufzeiten-Verteilung bei der statischen Analyse

Abbildung 19 zeigt die Laufzeiten-Verteilung der statischen Analyse. Zu sehen ist, dass die Laufzeit beim Großteil der Applikationen weniger als 20 Sekunden betrug. Die meisten Applikationen konnten sogar innerhalb der ersten 5 Sekunden analysiert werden.

Gesammelte Rohdaten

Bei der statischen Analyse werden unter anderem die in Tabelle 5 aufgelisteten Daten gesammelt. Viele der extrahierten Daten können dabei fast unverändert von *Androguard* übernommen werden. Einige Daten, beispielsweise die gesammelten `externalMethodCalls` und die `actualPermissions`, werden zuvor aufbereitet, um Speicher zu sparen (siehe Listing 4 und 5). Methoden-Aufrufe mit gleichen Signaturen werden dafür zusammengefasst und mit der Anzahl ihrer Vorkommen versehen. Die gesammelten Daten werden später zum Erstellen von Features sowie für die Steuerung der dynamischen Analyse genutzt.

Listing 4: Statische Analyse: Externe Methoden Aufrufe

```
...
"Landroid/os/Environment;": {
  "getExternalStorageDirectory": {
    "()Ljava/io/File;": NumberInt(3)
  }
}
...
```

Listing 5: Statische Analyse: Benötigte Berechtigungen

```
"READ_PHONE_STATE": {
  "Lorg/acra/util": {
    "Landroid/telephony/TelephonyManager;": {
      "getDeviceId": {
        "()Ljava/lang/String;": NumberInt(1)
      }
    }
  }
}
...
```

4.2.3 Modul: Dynamische Analyse

Die dynamische Analyse stellt den aufwendigsten Teil des Frameworks dar, nicht nur was den Entwicklungsaufwand, sondern auch was die Laufzeit betrifft. Besonderes Augenmerk liegt deshalb bei der dynamischen Analyse darauf, diese möglichst performant zu gestalten. Aus diesem Grund wird nicht, wie in vielen anderen Arbeiten (siehe 3.2.2 und 3.2.3), *QEMU* als Emulator eingesetzt, sondern auf *VirtualBox* und *Android-x86* gesetzt.

Warum VirtualBox?

Der Android Emulator in seiner Standard-Konfiguration ist sehr langsam. Allein der Start des Emulators und der Bootvorgang des Android-Systems können einige Minuten beanspruchen, was bei der dynamischen Analyse zu viel längeren Laufzeiten führen würde.

validApk	`True` oder `False`, je nachdem ob *Androguard* in der Lage war die APK-Datei zu öffnen oder nicht.
package	Das Applikations-Package beziehungsweise der Namensraum, in dem sich die Applikation befindet, beispielsweise `at.torghele.camera`.
mainActivity	Diejenige Aktivität, die beim Start einer Applikation über deren Icon im Launcher ausgeführt wird, beispielsweise `CameraActivity`.
activities	Alle weiteren Aktivitäten, über die eine Applikation verfügt. Diese werden später bei der dynamischen Analyse nacheinander gestartet, um eine möglichst gute Code-Abdeckung zu erreichen.
providers	Alle Content-Provider, die von einer Applikation zur Verfügung gestellt werden.
receivers	Alle Broadcast-Receiver, die eine Applikation enthält. Die entsprechenden Broadcasts werden während der dynamischen Analyse gesendet.
services	Alle Dienste, die eine Applikation zur Verfügung stellt. Diese werden zum Beginn der dynamischen Analyse gestartet.
libraries	Externe Bibliotheken, die eine Applikation benötigt, beispielsweise die Google Maps Bibliothek.
isCryptoCode	`True` oder `False`, je nachdem ob verschlüsselte Code-Teile gefunden wurden oder nicht.
isDynamicCode	`True` oder `False`, je nachdem ob Code dynamisch nachgeladen wird oder nicht.
isNativeCode	`True` oder `False`, je nachdem ob nativer Code vorhanden ist oder nicht.
isReflectionCode	`True` oder `False`, je nachdem ob Code Reflexion eingesetzt wird oder nicht.
permissions	Alle Berechtigungen, die von einer Applikation in deren Manifest-Datei angefordert werden.
externalMethodCalls	Enthält alle Aufrufe von Methoden, die sich außerhalb des eigenen Namensraumes befinden. Hier sind auch alle Android API Aufrufe enthalten (siehe Listing 4). Außerdem wird zu jeder Methoden-*Signatur* die Anzahl der Aufrufe gespeichert.
actualPermissions	Enthält alle Methoden, die eine bestimmte Berechtigung voraussetzen (siehe Listing 5). Auch hier wird zu jeder Methoden-*Signatur* die Anzahl der Aufrufe gespeichert.

Tabelle 5: AndroSOM: Statische Features

Es wäre damit aus Performance-Gründen auch nur sehr schwer möglich, die dynamische Analyse mittels `strace`, vor allem wenn damit der *Zygote*-Prozess und damit nahezu das gesamte System überwacht wird, durchzuführen. Aus diesem Grund verwenden einige andere Arbeiten (siehe 3.2.2 und 3.2.3) speziell entwickelte Kernel-Module zum Aufzeichnen von System-Aufrufen.

Es gibt mehrere Möglichkeiten, um den Android Emulator zu beschleunigen und somit eine Analyse mittels `strace` zu ermöglichen:

- Den Bootvorgang durch ein Snapshot beschleunigen / überspringen
- CPU Hardware-Beschleunigung
- GPU Hardware-Beschleunigung

Durch das Wiederherstellen eines Snapshots können zwei Probleme gelöst werden. Einerseits kann der Bootvorgang *übersprungen* werden, wodurch das System innerhalb von wenigen Sekunden anstatt mehreren Minuten zur Verfügung steht, andererseits steht für jeden Analyse-Durchlauf ein sauberes, unverändertes System zur Verfügung, wenn das System zuvor immer wieder auf diesen Snapshot zurückgesetzt wird. Sowohl *QEMU* als auch *VirtualBox* verfügen über die Möglichkeit, Snapshots zu erzeugen.

Die Standard-CPU-Architektur von Android für Smartphones und Tablet-PCs ist ARM. Wenn die ARM-Architektur auf einem x86-System emuliert werden muss, entsteht ein enormer Overhead. Deshalb stellt das Android SDK für API-Level 10 und 15+ auch x86-Images zur Verfügung. Diese können auch unter *QEMU* Hardware-beschleunigt ausgeführt werden. In *VirtualBox* können auf einem Host-Computer mit x86-Architektur nur x86 Android Images ausgeführt werden.

Neben der Emulation der CPU entstehen vor allem durch die Emulation der GPU bei der Darstellung des GUI Performance-Engpässe. Ab API-Level 14 kann unter *QEMU* auch dafür Hardware-Beschleunigung genutzt werden. Es entsteht dabei allerdings folgendes Problem: Snapshots und GPU Hardware-Beschleunigung schließen sich unter *QEMU* gegenseitig aus.

Um ein flüssig laufendes System für die dynamische Analyse zu erreichen, dem neben der Ausführung von Applikationen (darunter auch Spiele mit 3D-Inhalten) noch genügend Ressourcen zur Verfügung stehen, um problemlos `strace` auf nahezu allen laufenden Prozessen und `tcpdump` auszuführen, müssen CPU und GPU Hardware-beschleunigt sein. Wird eine der beiden Komponenten emuliert, läuft die AndroSOM Sandbox nicht mehr flüssig.

Unter *QEMU* müsste in diesem Fall auf Snapshots verzichtet werden, wodurch nicht nur der Bootvorgang bei jedem Durchlauf abgewartet, sondern auch sichergestellt werden müsste, dass ein unverändertes Android-System vorliegt. *VirtualBox* hingegen unterstützt alle drei Features. Mit einem x86-Image unter *QEMU* würde der Bootvorgang zwar von mehreren Minuten auf rund 90 Sekunden verkürzt, was allerdings bei 2691 getesteten Samples immer noch einer rund 2 1/2 Tage längeren Analyse-Laufzeit entspricht.

Android-x86

Für AndroSOM wurde unter Verwendung des *Android-x86* Projektes ein zum Zeitpunkt der Implementierung des Frameworks aktuelles Android 4.4.2 System-Image kompiliert. Dieses beinhaltet einige zusätzliche Werkzeuge, beispielsweise `tcpdump` und `busybox`[19].

Zum Erzeugen dieses Images wurde ein Build-Environment aufgesetzt. Zusätzlich zu den in der Dokumentation angegebenen Abhängigkeiten[20] mussten noch `squashfs-tools`, `gettext`, `yasm` und `javadoc` installiert werden, damit der Build-Vorgang erfolgreich abgeschlossen werden konnte.

Listing 6 zeigt die Build-Konfiguration. Wichtig ist hierbei, dass die `TARGET_BUILD_VARIANT` auf `eng` gesetzt ist. Andere Möglichkeiten wären `user` oder `userdebug`. Dann ständen allerdings für AndroSOM wichtige Funktionen nicht zur Verfügung. Handelt es sich um einen Eng(ineering)-Build, werden nahezu alle vorhandenen Module und Werkzeuge in das resultierende Android-Image, welches schließlich durch den Befehl `make iso_img` erstellt werden kann, integriert. Außerdem ist die Android Debug Bridge (ADB) standardmäßig aktiviert und das System *gerootet*. Das Builden des gesamten Android-Systems dauert, je nach System, mehrere Stunden.

Listing 6: buildspec.mk

```
1 TARGET_PRODUCT:=android_x86
2 TARGET_BUILD_VARIANT:=eng
3 TARGET_BUILD_TYPE:=release
4 TARGET_KERNEL_CONFIG:=android-x86_defconfig
5 CUSTOM_TARGETS += tcpdump
```

Nach dem Erstellen des Images wurde dieses in *VirtualBox* installiert. Da das *Android-x86* Repository aus Lizenzgründen keine Applikationen von Google, wie beispielsweise „Google Maps“ enthalten darf, wurden diese manuell installiert.

Die VM wurde nach Abschluss der Installation des Android-Systems und aller Standard Google-Applikationen sowie Bibliotheken als Open Virtualization Format (OVA)-Datei exportiert. Diese wird auf *Dropbox*[21] bereitgestellt und kann direkt über das AndroSOM GUI bezogen werden.

Funktionsweise der dynamischen Analyse

Abbildung 20 beschreibt den groben Ablauf der dynamischen Analyse. Durchgehende Pfeile entsprechen hierbei dem Fluss von Daten, während gestrichelte Pfeile andeuten, welche Komponente von welcher anderen Komponente gesteuert wird.

Wie bereits bei der statischen Analyse zuvor, wird auch hier überprüft, ob die benötigten *VirusTotal*-Metadaten vorhanden sind und damit entschieden, ob das Sample analysiert

19. http://www.busybox.net
20. https://source.android.com/source/initializing.html
21. https://www.dropbox.com/s/45hqamphm36mavw/AndroidVMFull.ova

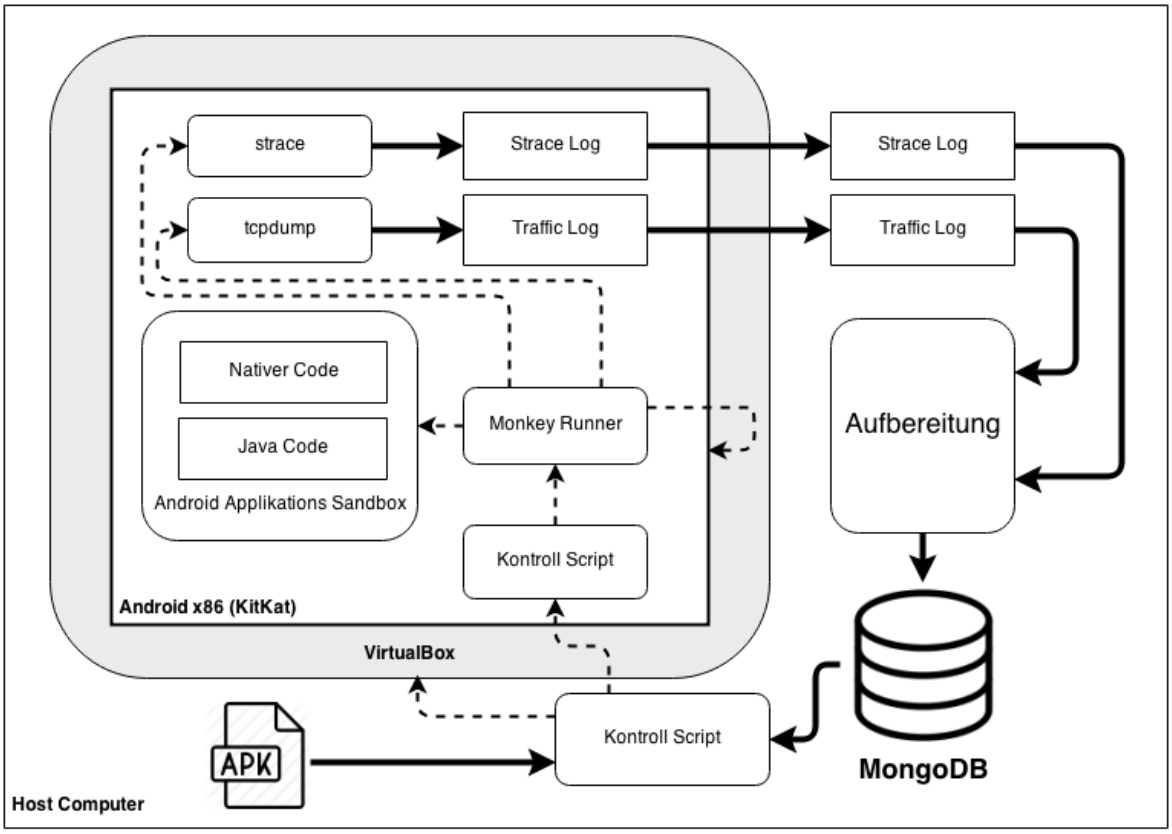


Abbildung 20: AndroSOM: Dynamische Analyse

werden soll oder nicht. Ein weiterer Grund für den vorzeitigen Abbruch der Analyse ist das Fehlen der statischen Analyse eines Samples. Aus dieser werden zumindest der Applikations-Namensraum sowie die Haupt-Aktivität benötigt, um die Applikation später in der AndroSOM Sandbox auszuführen.

Da es in vereinzelten Fällen (64 von 2691 analysierten Samples oder 2,38%) vorkam, dass das Android-System während der Ausführung eines Samples gänzlich einfror, wodurch die dynamische Analyse ohne manuelles Einschreiten niemals zu einem Ende gekommen wäre, wurde ein Timeout von 600 Sekunden festgelegt (300 Sekunden mehr, als ein normaler Analyse-Prozesses dauern sollte, siehe Abbildung 21). Dafür wird ein separater Thread gestartet, der nach Ablauf der gegebenen Zeit die Sandbox ordnungsgemäß herunterfährt und den Haupt-Thread beendet.

Die dynamische Analyse läuft dabei folgendermaßen ab:

1. **get_broadcasts():** Zuerst werden alle Broadcast-Receiver zusammen mit den System-Broadcasts, auf die sie hören, direkt aus dem Manifest der APK-Datei ausgelesen. Diese werden später, während der Laufzeit der Applikation, zur Generierung von Events genutzt.

2. **clean_state():** Durch diese Methode werden einerseits alle Dateien in `./tmp/`

gelöscht und andererseits der Befehl `VBoxManage snapshot AndroidVM restore cleanstate` aufgerufen. Dieser setzt die VM auf einen, während der Setup-Phase erstellten, sauberen Snapshot zurück. Das Zurücksetzen einer VM auf einen Snapshot dauert dabei zwischen 0,5 und 3 Sekunden. Damit herrschen für jeden dynamischen Analyse-Durchlauf die exakt gleichen Voraussetzungen.

3. **start_vm():** Wie der Name dieser Methode bereits vermuten lässt, wird an dieser Stelle mittels `VBoxManage startvm AndroidVM` die AndroSOM Sandbox gestartet. Da zuvor ein Snapshot wiederhergestellt wurde, in dem das Android-System bereits voll hochgefahren ist und auf BenutzerInnen Eingaben wartet, steht innerhalb von rund weiteren 2 Sekunden eine voll funktionsfähige Android Sandbox zur Verfügung. Zum Vergleich hätte der Bootvorgang eines in *QEMU* emulierten Android 4.4 ARM Systems auf derselben Hardware durchschnittlich rund 380 Sekunden beansprucht. Diese enorme Zeitersparnis kann erreicht werden, da der Bootvorgang durch das Snapshot gänzlich übersprungen werden kann und die Hardware-beschleunigte Ausführung des Android-Systems in einem VMM gegenüber einem Emulator um ein Vielfaches effizienter ist (siehe Kapitel 3.2.2).

4. **connect_adb():** Nachdem die VM gestartet wurde, wird die Android Debug Bridge ADB gestartet und eine Verbindung zum in der VM laufenden Android-System aufgebaut. Von nun an kann mittels des Befehls `adb shell` direkt auf die Android Kommandozeile zugegriffen werden. Es ist unter anderem auch möglich, mittels `adb push/pull` Dateien zu übertragen.

5. **push_tasks():** An dieser Stelle ist es nötig, das Shell-Script `./scripts/tasks.sh` auf das Android-System zu übertragen und diesem Ausführungsrechte zu gewähren. Dieses Script verfügt über folgende Funktionen, die direkt auf dem Android-System ausgeführt werden:

   - **fnPrepare():** Diese Funktion erstellt unter `/sdcard/features/` ein leeres Verzeichnis zum Sammeln von Log-Dateien, die später an den Host-Computer übertragen werden.
   - **fnLogpids():** Diese Funktion erwartet als Argument einen Android Username. Wenn diese Funktion aufgerufen wird, werden in einer Endlosschleife alle PIDs, die unter dem übergebenen Usernamen ausgeführt werden, aufgezeichnet. Diese Liste von PIDs wird zur Filterung der `strace`-Logs verwendet, um nur Prozesse zu berücksichtigen, die zur jeweiligen unter Beobachtung stehenden Applikation gehören.
   - **fnTcpdump():** Mit Hilfe dieser Funktion kann `tcpdump` zum Aufzeichnen des Netzwerk-Verkehrs auf dem Android-System gestartet oder gestoppt werden.
   - **fnStrace():** Mit Hilfe dieser Funktion kann `strace` zum Überwachen von System-Methoden auf dem Android-System gestartet oder gestoppt werden. `strace` wird dabei direkt auf den Zygote Prozess und all seine *geforkten* Unterprozesse angesetzt (siehe Kapitel 2.3.6). Damit werden bis auf vereinzelte

System-Prozesse alle Prozesse, die auf dem Android-System ausgeführt werden, überwacht. Für jede PID wird dabei eine eigene Log-Datei angelegt. Durch die Aufzeichnung der PIDs im Hintergrund mit Hilfe der zuvor beschriebenen `fnLogpids()` Funktion können alle Log-Dateien von anderen Applikationen im Nachhinein herausgefiltert werden.

- **fnTransfer():** Diese Funktion sendet alle Dateien, die sich im Log-Verzeichnis befinden, mittels File Transfer Protocol (FTP) an den Host-Computer. Der Umweg über FTP wurde aus Performance-Gründen anstelle von `adb pull` gewählt. Die Dauer des Datentransfers konnte so, abhängig vom jeweiligen Sample, von mehreren Minuten auf wenige Sekunden reduziert werden.
- **fnMonkey():** Diese Funktion erwartet als Argumente den Prozess-Namen, die gewünschte Anzahl an zu generierenden Events und einen Seed. Danach werden der entsprechende Prozess (die zu testende Applikation) gestartet und Events erzeugt. Durch Angabe eines Seeds sind diese Events nicht zufällig, sondern reproduzierbar.

6. **install_apk():** Diese Methode installiert das zu analysierende Sample mittels ADB in der AndroidSOM Sandbox. Dafür wird der Befehl `adb install <apk Pfad>` verwendet.

7. **get_username():** Eine Liste aller installierten Applikationen wird vom Android-System in einer Datei namens `packages.list` geführt. Neben dem eindeutigen Paket-Namen einer jeden Applikation können hier auch Informationen wie beispielsweise die dazugehörige UID ausgelesen werden. Es ist sehr einfach, von einer Android UID den dazugehörigen Username abzuleiten, da diese einem fixen Schema folgen. Die UID `1005` entspricht beispielsweise dem Username `u0_a5`, die vorangehende `100` wird also einfach durch `u0_a` ersetzt. Um den Username zu erhalten, welcher mit der zuvor installierten Applikation verknüpft ist, wird die `packages.list` mittels `adb pull` heruntergeladen und entsprechend geparst. Der Username, unter dem die zu analysierende Applikation ausgeführt wird, ist für die Aufzeichnung der PIDs durch `fnLogpids()` nötig.

8. **start_tcpdump():** Diese Methode startet mittels der ADB `tcpdump` in der AndroSOM Sandbox. Dafür wird `adb shell sh /data/local/tasks.sh tcpdump start` aufgerufen.

9. **start_strace():** Diese Methode startet mittels der ADB `strace` in der AndroSOM Sandbox. Dafür wird `adb shell sh /data/local/tasks.sh strace start` aufgerufen. Außerdem müssen ab diesem Zeitpunkt alle PIDs aufgezeichnet werden, die mit der zu analysierenden Applikation in Verbindung stehen. Dafür wird `adb shell sh /data/local/tasks.sh logpids <username>` gestartet.

10. **start_logcat():** Bei *Logcat* handelt es sich um ein Android-Werkzeug zum Mitlesen von verschiedenen Android-spezifischen Log-Dateien. Während der dynamischen Analyse wird das *Radio*-, *Events*- und *Main*-Log in entsprechende Dateien

am Host-Computer geschrieben und später gesichert. Diese könnten zur Fehlersuche herangezogen werden.

Ab diesem Zeitpunkt werden das Android-System, sowie die zu analysierende Applikation, in der Sandbox überwacht und die automatisierte Generierung von Events kann beginnen.

11. **send_broadcasts():** Diese Methode sendet mittels `adb shell am broadcast` die zuvor extrahierten Broadcasts direkt an ihre Empfänger. Damit sollen mögliches schädliches Verhalten ausgelöst oder Dienste gestartet werden.

12. **start_all_services():** Um sicherzustellen, dass alle in einer Applikation enthaltenen Dienste ausgeführt werden, wird zusätzlich mittels `adb shell am startservice` versucht, alle vorhandenen Dienste zu starten.

13. **run_all_activities():** Diese Methode startet nacheinander jede in der Applikation gefundene Aktivität und führt auf dieser anschließend den Android Monkey zum Simulieren von Events aus. Dabei werden 100 Events an jede Aktivität geschickt. Die Gesamt-Laufzeit dieser Methode wird aufgezeichnet.

14. **monkey(restzeit):** Da alle Samples bei der dynamischen Analyse gleich lange analysiert werden sollten und es von der Anzahl der Aktivitäten abhängt, wie lange `run_all_activities()` läuft, wird an dieser Stelle der Android Monkey so oft hintereinander ausgeführt, bis die vordefinierte Gesamt-Analyse-Zeit erreicht wurde. Sollte der Android Monkey hierfür mehr als 20 Anläufe brauchen, kann davon ausgegangen werden, dass die zu analysierende Applikation direkt nach deren Start abstürzt. In diesem Fall wird abgebrochen, um Zeit zu sparen.

15. **stop_strace():** Diese Methode beendet mittels der ADB `strace` in der AndroSOM Sandbox. Dafür wird `adb shell sh /data/local/tasks.sh strace stop` aufgerufen.

16. **stop_tcpdump():** Diese Methode beendet mittels der ADB `tcpdump` in der AndroSOM Sandbox. Dafür wird `adb shell sh /data/local/tasks.sh tcpdump stop` aufgerufen.

17. **pull_data():** Diese Methode startet einen minimalen *Python* FTP-Server auf dem Host-Computer. Anschließend wird `adb shell sh /data/local/tasks.sh transfer` aufgerufen und alle Dateien, die sich im Verzeichnis `/sdcard/features/` der AndroSOM Sandbox befinden, werden an den Host-Computer übertragen. Wenn alle Dateien übertragen wurden, kann der FTP-Server beendet werden.

18. **stop_vm():** Nachdem alle Daten übertragen wurden kann die VM-Instanz heruntergefahren werden. Dies geschieht mittels `VBoxManage controlvm AndroidVM poweroff`. Auch die ADB muss an dieser Stelle beendet werden um die nötigen Ports für den nächsten Durchlauf freizugeben.

An dieser Stelle werden alle gesammelten Daten aufbereitet und gesichert. Da das AndroSOM Framework derzeit nur über eine Sandbox-Instanz verfügt und nicht auf die dynamische Analyse von mehreren Samples gleichzeitig ausgelegt ist, kann die Analyse derzeit nicht parallelisiert werden.

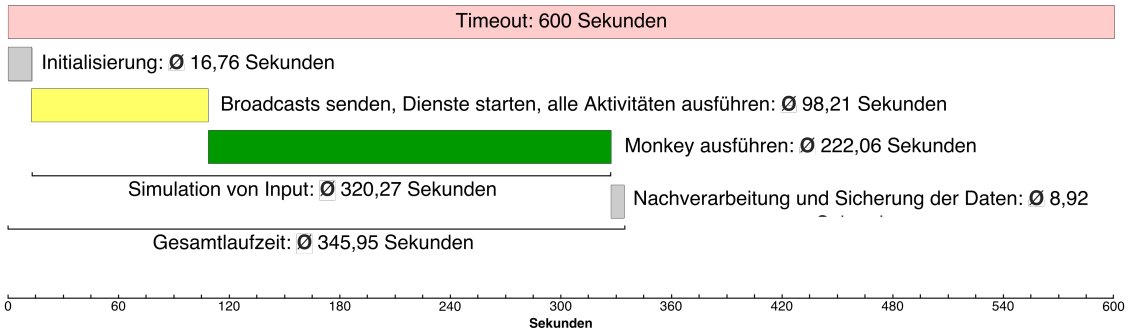


Abbildung 21: AndroSOM: Laufzeit der dynamischen Analyse

Abbildung 21 zeigt den zeitlichen Ablauf der dynamischen Analyse. Die Initialisierungs-Phase beinhaltet dabei alle oben genannten Punkte bis hin zum Punkt `start_logcat()`, an dem die VM hochgefahren, die zu untersuchende APK installiert und alle Aufzeichnungs-Werkzeuge gestartet wurden. Dieser Schritt dauert durchschnittlich `16,76` Sekunden. In der zweiten Phase wird versucht, möglichst viele Ausführungs-Pfade der zu untersuchenden Applikation zu durchlaufen. Dafür werden *Broadcasts* gesendet, Dienste gestartet und jede in der Manifest-Datei der Applikation gefundene Aktivität gestartet und mit Zufalls-Events durch den Android *Monkey* stimuliert. Diese zweite Phase dauert durchschnittlich `98,21` Sekunden und ist gelb markiert. In der nun folgenden Phase der dynamischen Analyse wird der Android *Monkey* so lange ausgeführt, bis die vordefinierte Mindest-Analyse-Zeit von `300` Sekunden überschritten wurde. Diese Phase ist grün markiert und wird durchschnittlich nach weiteren `222,06` Sekunden beendet. In der letzten Phase der dynamischen Analyse werden die Aufzeichnungs-Werkzeuge gestoppt, die gesammelten Daten aus der VM heruntergeladen, die VM beendet und anschließend die Daten aufbereitet, was durchschnittlich weitere `8,92` Sekunden beansprucht. Die Gesamt-Laufzeit eines dynamischen Analyse-Durchlaufes beträgt somit durchschnittlich 345,95 Sekunden, was bei 2691 analysierten Samples einer Laufzeit von fast 11 Tagen entspricht. Die gemessenen Durchschnitts-Zeiten beruhen auf den Daten von erfolgreichen Analyse-Durchläufen. Durchläufe, die nach dem gesetzten Timeout von `600` Sekunden beendet wurden, wurden hierfür nicht berücksichtigt.

Abbildung 22 zeigt die Laufzeiten-Verteilung bei der dynamischen Analyse. Zu sehen ist, dass die Analyse der meisten Samples zwischen 300 und 400 Sekunden in Anspruch genommen hat. Der einzelne Ausschlag bei 600 Sekunden wurde durch diejenigen Applikationen verursacht, die nach dem Erreichen des Timeouts vom Framework beendet wurden.

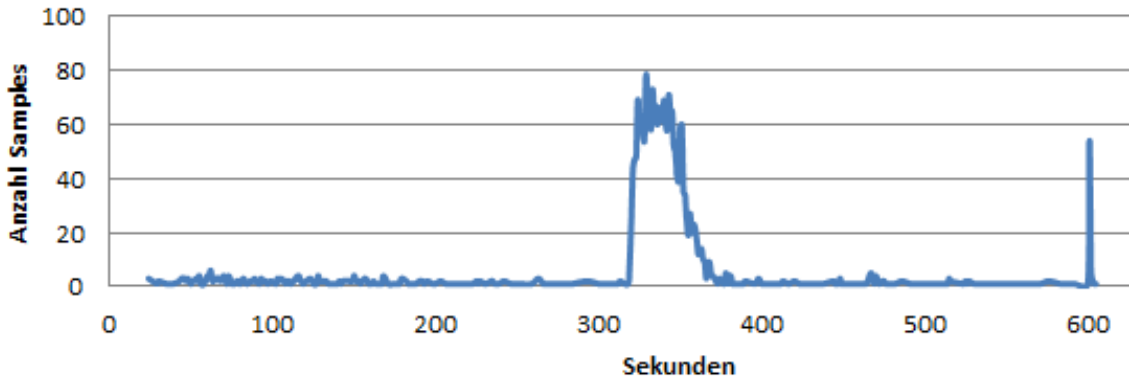


Abbildung 22: Laufzeiten-Verteilung bei der dynamischen Analyse

valid	`True` oder `False`, je nachdem ob die dynamische Analyse erfolgreich war oder nicht. Die dynamische Analyse schlägt fehl, wenn bei der Ausführung der Applikation ein vordefiniertes Zeitlimit überschritten wird oder die zu testende Applikation immer gleich nach dem Start abstürzt.
chmodedFiles	Alle Dateien, deren Zugriffs- und Ausführungs-Berechtigungen durch die Applikation während der Laufzeit verändert wurden, zusammen mit der Anzahl der Zugriffe.
accessedFiles	Alle Dateien, deren Zugriffs- oder Ausführungs-Berechtigungen durch die Applikation während der Laufzeit getestet wurden, zusammen mit der Anzahl der Tests.
openedFiles	Alle Dateien, auf die von der Applikation zur Laufzeit lesend oder schreibend zugegriffen wurde, zusammen mit der Anzahl der Zugriffe.
methodCalls	Alle Aufrufe von System-Methoden wie beispielsweise `clock_gettime`, `mprotect` oder `getuid32` zusammen mit der genauen Anzahl dieser Aufrufe.

Tabelle 6: AndroSOM: Dynamische Features

Gesammelte Rohdaten

Bei der dynamischen Analyse eines Samples werden einerseits Aufrufe von System-Methoden und der Zugriff auf Dateien durch `strace`, andererseits der Netzwerk-Verkehr durch `tcpdump` aufgezeichnet. Die Ergebnisse werden in separaten *MongoDB* Collections abgelegt und enthalten unter anderem die in den Tabellen 6 und 7 aufgeführten Daten beziehungsweise Features. Alle Features aus der dynamischen Analyse, die in Tabelle 6 aufgelistet sind, können eindeutig der zu untersuchenden Applikation zugeschrieben werden. Die Features der Analyse des Netzwerk-Verkehrs in Tabelle 7 betreffen das gesamte System, könnten also auch von anderen Applikationen oder dem Android-System selber stammen. Da für jeden Durchlauf dieselben Voraussetzungen herrschen, sollte das allerdings kein Problem darstellen.

accessedIps	Enthält alle Internet Protocol (IP) Adressen, auf die während der Laufzeit der Applikation zugegriffen wurde, zusammen mit der Anzahl der Zugriffe.
accessedHostnames	Enthält alle Hostnames, auf die während der Laufzeit der Applikation zugegriffen wurde, zusammen mit der Anzahl der Zugriffe.
destinationPorts	Enthält alle Ziel-Port-Nummern, über die Netzwerk-Aktivitäten während der Laufzeit der Applikation stattfanden, zusammen mit der Anzahl dieser Aktivitäten.
usedHttpMethods	Enthält, wenn vorhanden, die Anzahl aller getätigten HTTP-Requests (`GET`, `POST`, `HEAD`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`)
responseCodes	Enthält die zurückgegebenen Response-Codes samt deren Häufigkeit.
contentTypes	Enthält die Content-Types, beispielsweise `application/binary`, zusammen mit deren Anzahl.

Tabelle 7: AndroSOM: Traffic Features

4.3 Nachverarbeitung und Generierung der Datensätze

Wie bereits zuvor erwähnt, erfolgt die Generierung der Datensätze in zwei Schritten. Zuerst werden alle Features aus der statischen und dynamischen Analyse zusammengetragen und zusammen mit Metadaten in eine MongoDB Collection geschrieben.

Listing 7: Nachverarbeitung: Features mit Metadaten

```
1  {
2     "_id": "static_externalMethodCalls__org/xmlpull/v1/XmlPullParser->
          setInput",
3     "inMalwareCount": NumberInt(297),
4     "inBenignCount": NumberInt(833),
5     "maxValue": NumberInt(42),
6  }, {
7     "_id": "traffic_responseCodes__304",
8     "inMalwareCount": NumberInt(33),
9     "inBenignCount": NumberInt(127),
10    "maxValue": 150.55299757169,
11 }
```

Wie in Listing 7 zu sehen ist, wird für jedes Feature eine eindeutiger Identifikator (`_id`) generiert. Metadaten, die dazu abgespeichert werden, sind der `inMalwareCount`, welcher angibt in wie vielen Schadsoftware-Samples dieses Feature gefunden wurde, der `inBenignCount`, welcher angibt, in wie vielen legitimen Applikationen dieses Feature gefunden wurde und der Maximal-Wert, der beschreibt, wie häufig dieses Feature maximal in einem Sample, in dem es enthalten war, gefunden wurden. Da es bei der Laufzeit der dynamischen Analyse immer geringfügige Abweichungen gibt, wird dieser Maximal-Wert auf 60 Sekunden normalisiert.

Im zweiten Schritt werden mit Hilfe der zuvor generierten Feature-Collection Sample-spezifische Feature Vektoren erstellt. Diese werden anschließend in verschiedene Datensätze geschrieben. Dabei kann über das GUI für die verschiedenen Features ein Filter gesetzt werden, welcher angibt, in wie vielen Samples ein bestimmtes Feature vorhanden sein muss, um in die Datensätze aufgenommen zu werden.

Alle Gewichte der generierten Feature-Vektoren, sofern es sich nicht um boolesche Werte handelt, werden, wie in Listing 8 zu sehen ist, für die spätere Verwendung in *MatLab* auf den Wertebereich $[0, 1]$ normalisiert.

Listing 8: AndroSOM Feature-Vektor

```
0.0092092825564, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.0123017112
748, 0, 0, 0.0555065764217, 0.0555065764217, 0.00900641305601, 0, 0,
0, 0, 0.01230171127481, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, 0.2854862
6368457, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0.6515299891291, 0, 0, 0, 0, 0, 0, 0, 0, 0.005509673730331, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
```

Listing 8 zeigt, wie ein Feature-Vektor aufgebaut sein kann. Dieser kann beispielsweise Daten aus der statischen Analyse (gelb markiert), aus der dynamischen Analyse von System-Aufrufen (grün markiert) und aus der Analyse des Netzwerk-Verkehres während der dynamischen Analyse (blau markiert) enthalten.

Nachdem die verschiedenen Feature-Vektoren für alle Samples erzeugt wurden, können daraus die Datensätze für *MatLab* generiert werden.

5 Resultate

In diesem Kapitel werden anfangs die verwendeten APK-Datensätze beschrieben, sowie Einblicke in diese gegeben. Anschließend werden die Messmethode beschrieben und Messergebnisse für verschiedene CP-ANN-Modelle präsentiert. Abschließend werden die Top-10 CP-ANN-Modelle identifiziert, welche einer vierfachen Kreuz-Validierung unterzogen werden. Damit soll deren Leistung (*Accuracy*) in Hinblick auf zuvor unbekannte Samples gezeigt werden.

5.1 Applikations-Datensätze

Für diese Arbeit wurden sowohl schädliche als auch legitime Applikationen verwendet. Diese stammen aus bekannten Schadsoftware-Sammlungen, beziehungsweise wurden sie vom Autor dieser Arbeit aus dem offiziellen *Google Play Store* manuell heruntergeladen.

Schadsoftware

Die im Zuge dieser Arbeit verwendeten Android Schadsoftware-Samples stammen aus dem *Android Malware Genome Project* (Zhou und Jiang 2012), sowie von Contagio Mobile[22]. Das *Android Malware Genome Project* entstand im Zuge der Arbeit von Zhou and Jiang und beinhaltet 1260 Schadsoftware-Samples, die in 49 Familien eingeteilt werden können. Dieser Datensatz wurde zwischen August 2010 und Oktober 2011 aufgebaut. Die Contagio Mobile Datenbank beinhaltet rund 380 Schadsoftware Samples, die zwischen September 2012 und September 2014 gesammelt wurden. Insgesamt konnten mit Hilfe von *VirusTotal* 78 verschiedene Schadsoftware-Familien identifiziert werden.

Legitime Applikationen

Die legitimen Applikationen wurden zwischen dem 16. und 20. Februar 2014 vom Autor dieser Arbeit manuell aus dem offiziellen *Google Play Store* heruntergeladen. Dabei handelt es sich jeweils um die Top 60 Applikationen (auf Österreich bezogen) aus 34 Kategorien. Diese teilen sich auf 26 allgemeine und 8 Kategorien für Spiele auf. Insgesamt wurden dabei 2040 Applikationen gesammelt, wobei manche in mehreren Kategorien enthalten waren. Samples, die öfter als ein Mal vorhanden sind, werden nach der ersten statischen beziehungsweise dynamischen Analyse nicht mehr weiter berücksichtigt.

	Schadsoftware	Legitime Applikationen
ursprünglich	1640	2040
gefiltert	1080	1105

Tabelle 8: Ursprüngliche und gefilterte Samples

22. http://contagiominidump.blogspot.nl

Tabelle 8 zeigt die Anzahl von gesammelten Applikationen im Vergleich mit der Anzahl später tatsächlich verwendeter Samples. Herausgefiltert wurden mehrfach enthaltene Applikationen sowie Samples, die nicht eindeutig genug durch *VirusTotal* als Schadsoftware beziehungsweise legitime Applikationen klassifiziert werden konnten. Auch einige mutmaßlich legitime Applikationen aus dem offiziellen Google Play Store wurden entfernt, da sie zumindest bei einigen der 50 Viren-Scanner, die von *VirusTotal* unterstützt werden, als Schadsoftware identifiziert wurden.

Von *VirusTotal* als Schadsoftware erkannte Applikationen

Bei *VirusTotal* werden Samples mit 50 Viren-Scannern auf Schadsoftware überprüft. Nicht alle Viren-Scanner erkennen dieselben Samples, was Abbildung 23 verdeutlichen soll.

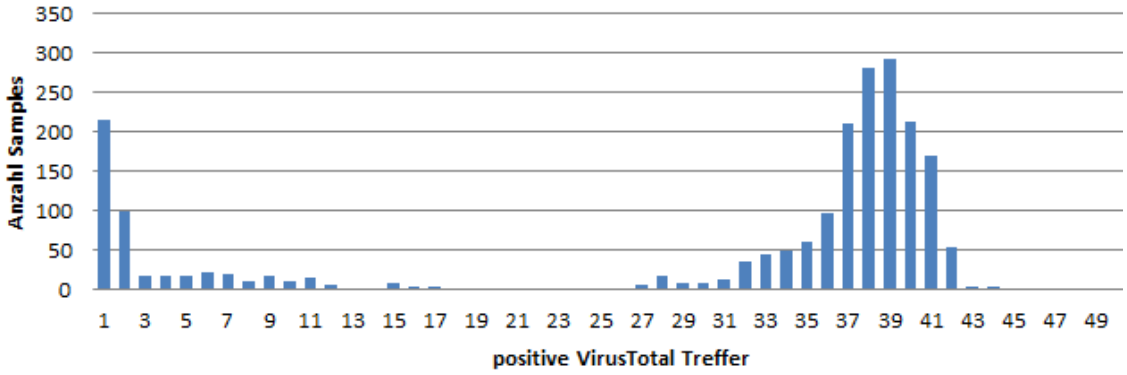


Abbildung 23: Als Schadsoftware erkannte Applikationen

Wie in Abbildung 23 zu sehen ist, wurden 215 Samples von nur einem beziehungsweise 100 Samples von genau zwei Viren-Scannern erkannt. Dabei handelt es sich vermutlich um *False Positives* (siehe Tabelle 10). Danach bewegt sich die Erkennungs-Rate zwischen 0 und 20 Samples, die von einer bestimmten Anzahl an Viren-Scannern als Schadsoftware klassifiziert wurden. Die Anzahl an als schädlich klassifizierten Samples steigt erst ab 32 positiven *VirusTotal*-Treffern merklich an und erreicht bei 39 verschiedenen Viren-Scannern, die 293 Samples als Schadsoftware identifiziert haben, ihren Höhepunkt. Keine der Applikationen wurde von 47 oder mehr Viren-Scannern erkannt.

Auswahl des Schwellenwertes für Schadsoftware

Um nicht zu viele Samples aus dem APK-Datensatz zu entfernen, musste ein geeigneter Schwellenwert für die Anzahl an Viren-Scannern festgelegt werden, die nötig sind, um ein Sample eindeutig als schädlich zu klassifizieren. Nur wenn ein Sample diesen Schwellenwert erreicht, wird es in dieser Arbeit als Schadsoftware-Sample weiterverwendet. Legitime Applikationen dürfen hingegen von keinem einzigen Viren-Scanner erkannt werden, um verwendet zu werden.

Abbildung 24 zeigt in Abständen von fünf Viren-Scannern, wie viele Samples zusätzlich aus der Wertung fallen, wenn ein bestimmter Schwellenwert gewählt wird.

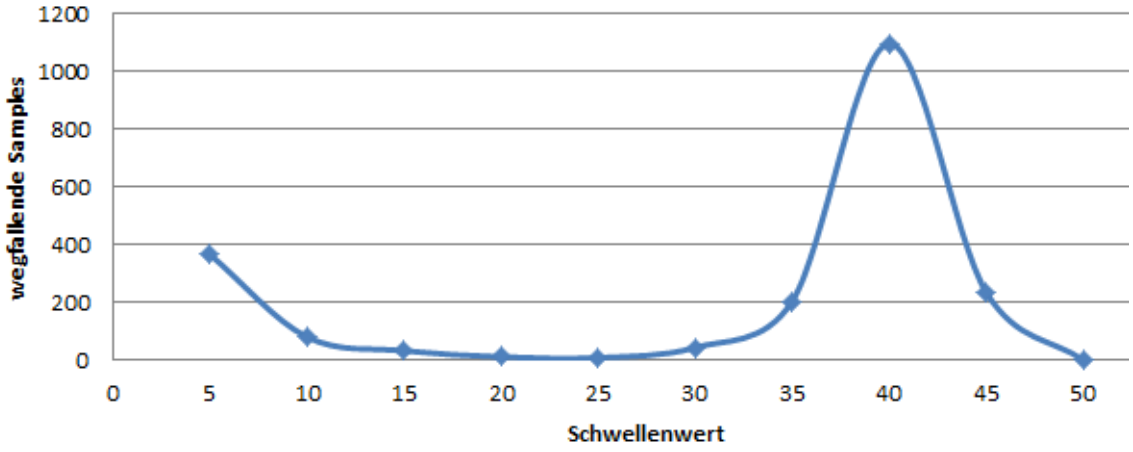


Abbildung 24: Schwellenwert für die Klassifizierung als Schadsoftware

5.1.1 Einblicke

In schädlichen sowie in legitimen Applikationen kommen bestimmte Features (siehe Tabelle 5, 6 und 7) unterschiedlich oft vor. Aus diesem Grund können schädliche von legitimen Samples auch unterschieden werden. Es folgt eine Auflistung der jeweils 25 am häufigsten in Schadsoftware beziehungsweise in legitimen Applikationen gefundenen Features.

Statische Analyse

Abbildung 25 zeigt eine Auflistung der 25 am häufigsten aus legitimen Applikationen extrahierten Features, die durch statische Analyse gewonnen wurden, im Vergleich zu den 25 am häufigsten in Schadsoftware gefundenen Features aus dieser Analyse-Art. Einige Features, welche möglicherweise auf Schadsoftware hindeuten, wurden rot markiert. Zu sehen ist die Initialisierung von *Intents* (siehe Kapitel 2.2.6), welche genutzt werden können, um Aktivitäten zu starten, Broadcasts zu senden sowie um Dienste zu starten beziehungsweise mit diesen zu kommunizieren. Schadsoftware kann *Intents* beispielsweise zum Starten von schädlichen Hintergrund-Diensten nutzen. Auch sehr oft kommen Methoden zum Manipulieren von Zeichenketten vor, welche beispielsweise für einfache Verschleierungs-Techniken (siehe Kapitel 3.2.1) genutzt werden können. Auch die Methode `getSystemService()` scheint unter den Top 25 Schadsoftware-Features auf. Über diese Methode kann beispielsweise ein `TelephonyManager`[23] instantiiert werden, welcher Zugang auf alle Telefonbuch-Daten bieten kann. Die Methode `currentTimeMillis()` könnte beispielsweise genutzt werden, um schädlichen Code erst nach einer bestimmten Laufzeit einer Applikation auszuführen und somit eine dynamische Erkennung in einer *Sandbox* zu erschweren.

23. http://developer.android.com/reference/android/telephony/TelephonyManager.html

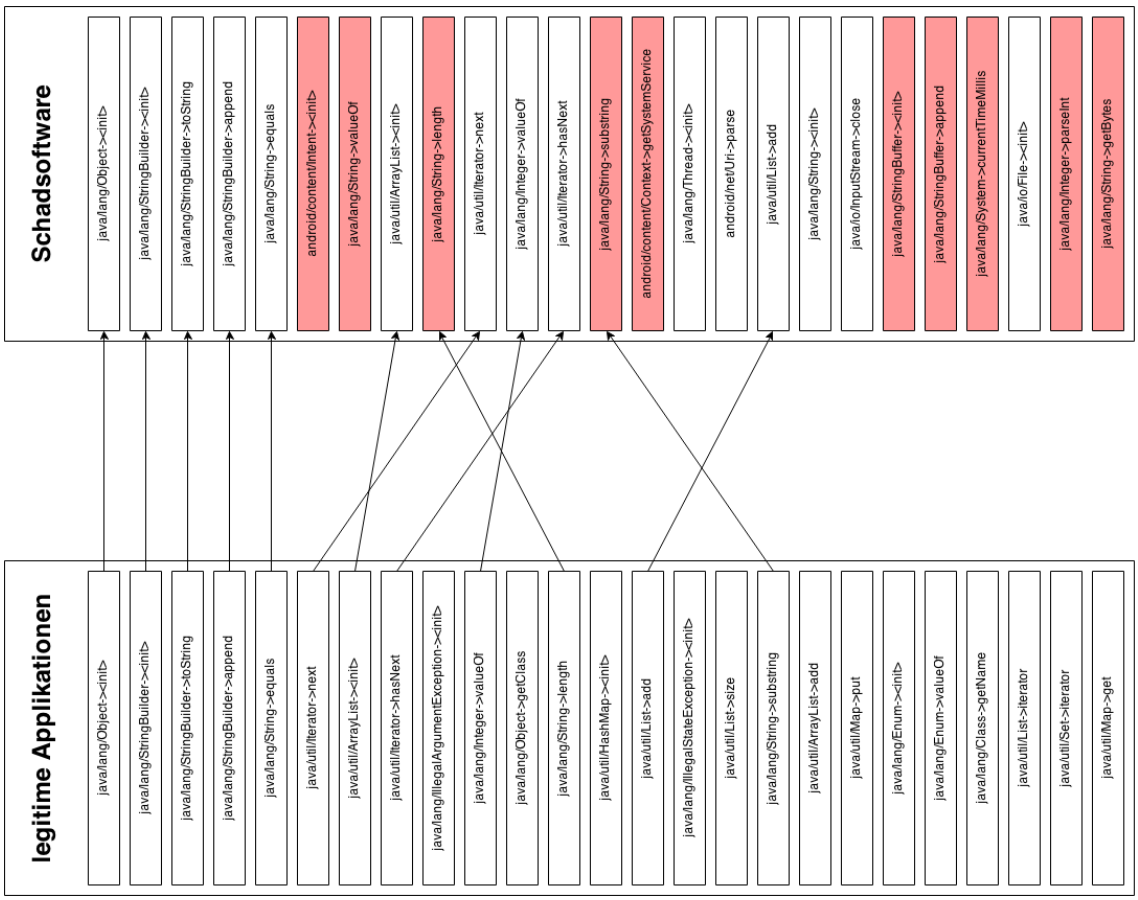


Abbildung 25: Vergleich der am häufigsten gefundenen statischen Features

Dynamische Analyse

Abbildung 26 zeigt eine Auflistung der 25 am häufigsten aus legitimen Applikationen extrahierten Features, die durch die dynamische Analyse von System-Aufrufen und Datei-Zugriffen gewonnen wurden, im Vergleich zu den 25 am häufigsten in Schadsoftware gefundenen Features. Einige Features, auf die näher eingegangen wird, da diese auf Schadsoftware hindeuten können, wurden rot markiert. Der `access` System-Aufruf liefert Informationen darüber, ob der aktuelle Prozess über die nötigen Berechtigungen verfügt, die zum Lesen, Schreiben oder Testen, ob eine Datei vorhanden ist, erforderlich sind. Eine Applikation kann damit feststellen, ob sie Zugriff auf System-Dateien oder -Partitionen hat (beispielsweise wenn es sich um ein *gerootetes* System handelt) und gegebenenfalls weitere Schritte einleiten, um Schaden anzurichten. Der System-Aufruf `fcntl64` wird zur Manipulation von Datei-Deskriptoren eingesetzt, vor allem aber um Dateien zu sperren, wenn auf diese beispielsweise gerade schreibend zugegriffen wird. Dieser System-Aufruf sowie `read` deuten auf erhöhte Lese- beziehungsweise Schreib-Aktivität am Filesystem hin, was für Schadsoftware sprechen kann. Über den System-Aufruf `capset` können einem Prozess bestimmte Berechtigungen gewährt werden. Der System-Aufruf `mount` könnte benutzt werden, um die normalerweise schreibgeschützte Android System-Partition mit Schreib-

zugriff neu einzubinden. Bei `ashmem` handelt es sich um das Android Shared-Memory Subsystem, durch welches die Binder-IPC umgangen werden kann.

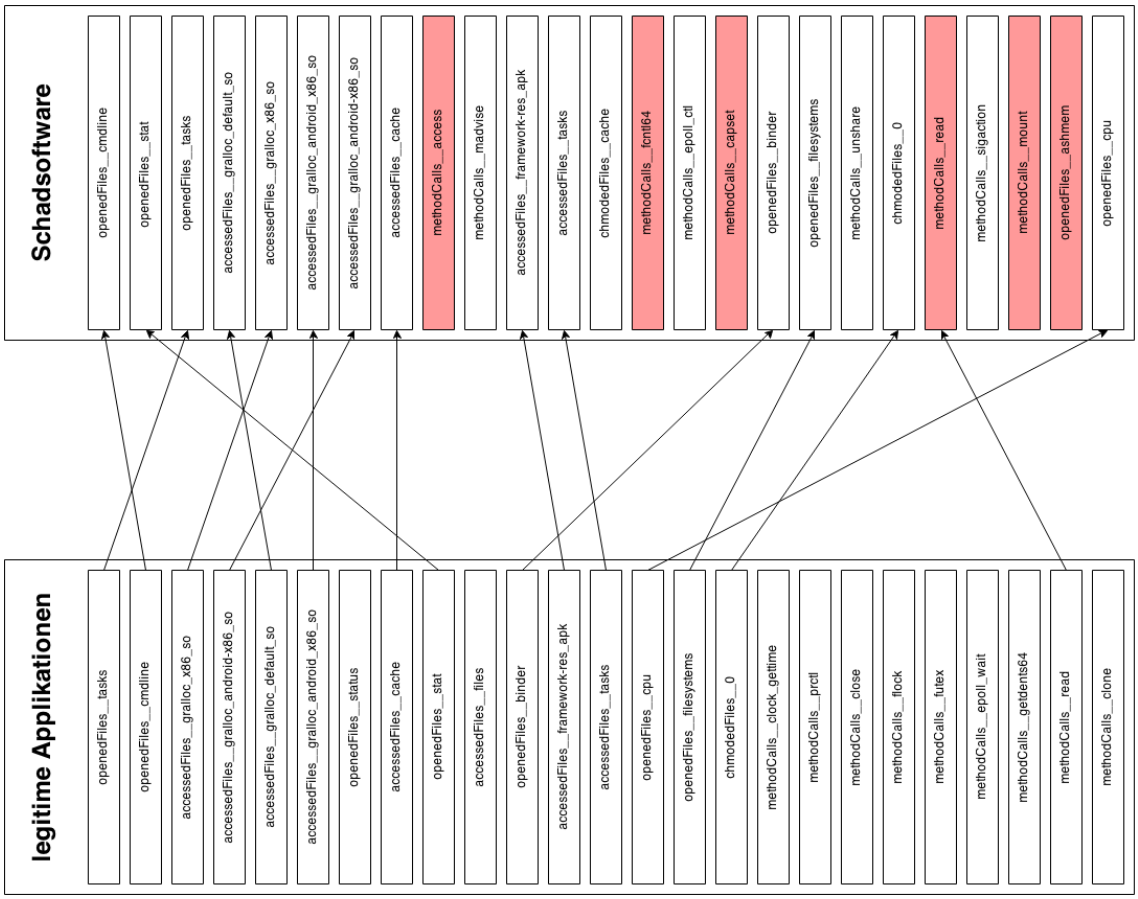


Abbildung 26: Vergleich der am häufigsten gefundenen dynamischen Features

Netzwerkverkehr Analyse

Abbildung 26 zeigt eine Auflistung der 25 am häufigsten aus legitimen Applikationen extrahierten Netzwerkverkehrs-Features, die während der dynamischen Analyse gesammelt wurden, im Vergleich zu den 25 am häufigsten in Schadsoftware gefundenen Features aus der Analyse des Netzwerkverkehrs. Einige Features, welche möglicherweise auf Schadsoftware hindeuten, wurden rot markiert. Auffällig ist hierbei, dass besonders oft der Response-Code `404` zurückgegeben wird. Grund hierfür könnte sein, dass bestimmte *Botnetz-Kontroll-Server* über die Zeit entdeckt wurden und offline gingen. Die verwendete Schadsoftware kommunizierte auch oft mit *Adwo*[24], einem chinesischen Werbenetzwerk beziehungsweise *adtouchnetwork.net* (offline). Legitime Applikationen hingegen verwendeten meist die Google-eigenen Werbenetzwerke *DoubleClick*[25] und *AdMob*[26]. Auch wird

24. http://www.adwo.com
25. http://www.google.com/doubleclick/
26. http://www.google.com/ads/admob/

oft eine Verbindung auf Port `8080` aufgebaut, was ungewöhnlich erscheint. Ebenfalls unter den Top 25 Features befinden sich vier IP-Adressen, mit denen nur Schadsoftware-Samples Verbindungen aufbauten.

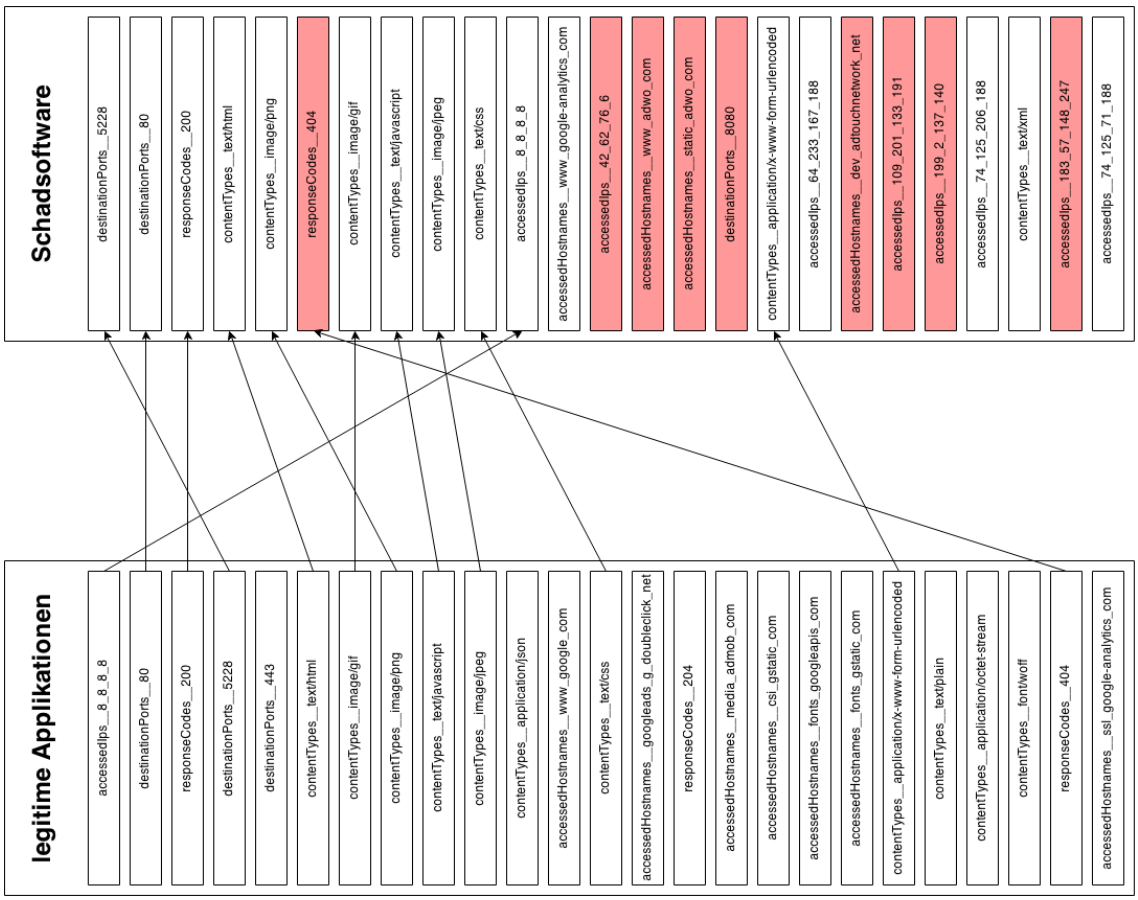


Abbildung 27: Vergleich der am häufigsten gefundenen Netzwerkverkehr Features

5.2 Messmethode

Um die Messungen auszuwerten, wurde *MatLab* verwendet. *MatLab* ist sowohl eine höhere Programmiersprache als auch eine interaktive Entwicklungsumgebung für numerische Berechnungen und Visualisierungen. *MatLab* kann dabei nicht nur zur Analyse von Daten und zur Algorithmen-Entwicklung, sondern auch zur Erstellung von Modellen und Anwendungen verwendet werden[27].

Unter *MatLab* ist es möglich, aus einzelnen Skripten oder Funktionen sogenannte *Toolboxes* zu erstellen. Diese sind meist anwendungsorientiert und können beispielsweise alle nötigen Werkzeuge enthalten, um ANNs zu berechnen.

27. http://de.mathworks.com/products/matlab/

In dieser Arbeit werden CP-ANNs mit Hilfe der *Kohonen und CPANN Toolbox* (Ballabio, Consonni und Todeschini 2009) berechnet. Es handelt sich dabei um eine Sammlung von *MatLab* Modulen und Funktionen, die speziell für die Berechnung von Kohonen Karten sowie CP-ANNs zur Klassifikation von Daten durch die „Milano Chemometrics and QSAR Research Group“ entwickelt wurde.

Um verschiedene CP-ANN Modelle zu berechnen, wurde folgendes *MatLab* Script verwendet:

Listing 9: MatLab Script zum Berechnen von CP-ANN-Modellen

```
function cpann(dataset_path)

X_full    = csvread(strcat(dataset_path,'/full_X'));
X_dynamic = csvread(strcat(dataset_path,'/dynamic_X'));
X_static  = csvread(strcat(dataset_path,'/static_X'));
X_traffic = csvread(strcat(dataset_path,'/traffic_X'));
classes   = csvread(strcat(dataset_path,'/class'));

i = 2;
while i<=16
    parfor j = 1:8
        settings = som_settings('cpann');
        settings.nsize = i;
        settings.epochs = j*25;
        tic;
        model = model_cpann(X_full,classes,settings);
        write_results(settings.nsize, settings.epochs, toc, model);
        ...
    end
    i = i + 2;
end
```

Listing 9 zeigt, wie verschiedene CP-ANN Modelle mit Hilfe eines *MatLab* Skripts automatisiert berechnet werden. Dazu wird der Funktion `cpann` der Pfad zu den mit Hilfe des implementierten „Feature Miners“ generierten Datensätzen übergeben, welche anschließend nacheinander in Matrizen ($[n \times p]$, n Samples, p Features) geladen werden. In den Vektor ($[n \times 1]$) `classes` wird für jedes Sample die richtige Klasse (1 = legitime Applikation, 2 = Schadsoftware) geladen.

Danach wird in einer doppelten Schleife mittels `model = model_cpann(X_*, classes, settings)` jeweils ein CP-ANN-Modell mit einer *Grid-Size* von $[2, 4, 6, 8, 10, 12, 14, 16]$ und $[25, 50, 75, 100, 125, 150, 175, 200]$ Epochen für jedes Dataset (`full_X`, `dynamic_X`, `static_X`, `traffic_X`) berechnet. Das entspricht einer totalen Anzahl von 256 berechneten CP-ANN-Modellen, wobei die Berechnung von einzelnen Modellen, je nach enthaltener Feature-Anzahl, *Grid-Size* und Epochen, mehrere Stunden in Anspruch nehmen kann.

Tabelle 9 zeigt einen Überblick über die für die Erstellung der CP-ANN-Modelle verwendeten Einstellungen.

Setting	Wert	Beschreibung
net_type	'cpann'	Der Typ des ANNs. Es kann zwischen `kohonen`, `cpann`, `skn` und `xyf` gewählt werden.
nsize	variabel	Die Anzahl der Neuronen für jede Seite der Karte. Handelt es sich um eine quadratische Karte (`topol = 'square'`), ergibt sich aus dem Quadrat dieses Wertes die Anzahl der vorhandenen Neuronen. Dieser Wert wird für jedes Modell verändert.
epochs	variabel	Dieser Wert definiert, wie oft hintereinander die Samples dem ANN zum Trainieren übergeben werden. Dieser Wert wird laufend verändert.
topol	'square'	Dieser Wert beschreibt die Form eines jeden Neurons. Neuronen können entweder eine quadratische oder hexagonale Form aufweisen.
bound	'toroidal'	Dieser Wert gibt an, dass jede Kante der Karte mit der gegenüberliegenden Kante verbunden ist. Dadurch besitzt jedes Neuron dieselbe Anzahl von benachbarten Neuronen.
training	'batch'	Dieser Wert bestimmt den Trainings-Algorithmus (*sequenziell* oder *batch*). Beim sequenziellen Training wird dem ANN jeweils nur ein Sample übergeben und die Gewichte entsprechend dem gewinnenden Neuron angepasst. Beim Batch-Training werden dem ANN alle Samples auf einmal übergeben und die Gewinner-Neuronen ermittelt. Die Gewichte werden anschließend auf Grundlage aller Gewinner-Neuronen angepasst.
init	'random'	Dieser Wert gibt an, wie die Gewichte vor dem Training initialisiert werden sollen. Gewichte können zufällig mit Werten von $0,1$ bis $0,9$ oder auf Basis des Eigenvektors, entsprechend der beiden größten Hauptkomponenten der jeweiligen Samples, initialisiert werden.
a_max	0.5000	Dieser Wert entspricht der initialen Lern-Rate.
a_min	0.0100	Dieser Wert entspricht der finalen Lern-Rate.

Tabelle 9: Verwendete CP-ANN-Einstellungen

Um die Berechnungen zu beschleunigen, wurde die *Parallel Computing Toolbox*[28] verwendet, was an der `parfor`-Schleife zu erkennen ist. Die Berechnung aller Modelle mit 4 parallel arbeitenden lokalen *Workern* nahm rund 8 Tage in Anspruch.

28. http://de.mathworks.com/products/parallel-computing

	als positiv erkannt	als negativ erkannt
echt positiv	TP	FN
echt negativ	FP	TN

Tabelle 10: Binäre Confusion Matrix (Sokolova, Japkowicz und Szpakowicz 2006)

5.3 Verwendete Metriken

Da es sich bei den Modellen um CP-ANNs handelt, und damit überwachtes Lernen ermöglicht wird, kann deren Leistung mittels verschiedener Metriken überprüft werden. Diese Metriken bauen auf einer sogenannten *Confusion Matrix* auf, in der korrekt und inkorrekt erkannte Samples pro Klasse aufgeschlüsselt sind.

Tabelle 10 zeigt eine *Confusion Matrix* für binäre Klassifikation. Dabei steht `TP` für „true positive“ und beinhaltet die Anzahl der positiven Samples, die auch als positiv klassifiziert wurden. `FP` steht für „false positive“ und beinhaltet die Anzahl der negativen Samples, die fälschlicherweise als positiv klassifiziert wurden. `FN` steht für „false negative“ und enthält die Anzahl der positiven Samples, die fälschlicherweise als negativ klassifiziert wurden. `TN` steht für „true negative“ und enthält die Anzahl der negativen Samples, die auch als negativ klassifiziert wurden (Sokolova, Japkowicz und Szpakowicz 2006).

Die Anzahl der richtig klassifizierten Samples kann hierbei der Haupt-Diagonale entnommen werden, während es sich bei allen anderen Elementen um die Anzahl der jeweils falsch klassifizierten Samples handelt. Im Falle einer perfekten Klassifizierung sind alle Elemente, die sich nicht auf der Haupt-Diagonale befinden, 0 und alle Elemente der Haupt-Diagonale entsprechen der Anzahl der Samples der jeweiligen Klasse (Frank und Todeschini 1994).

Error Rate

Die *Error Rate* (siehe Gleichung 6) beschreibt die prozentuale Häufigkeit von falsch klassifizierten Samples (auch *Falschklassifikations-Rate*).

$$ER = \frac{falsch\ klassifiziert}{alle\ Samples} = \frac{FP + FN}{TP + TN + FP + FN} \quad 0 \leq ER \leq 1 \tag{6}$$

Das Ergebnis liegt dabei im Wertebereich zwischen 0 und 1 (Frank und Todeschini 1994).

Accuracy

Das am häufigsten verwendete empirische Maß, um die Leistung eines gelernten Modelles zu messen, ist laut (Sokolova, Japkowicz und Szpakowicz 2006) die *Accuracy* (siehe Gleichung 7). Die *Accuracy* berücksichtigt dabei nicht einen möglichen Unterschied bei der Anzahl von Samples in den jeweiligen Klassen.

$$AC = \frac{richtig\ klassifiziert}{alle\ Samples} = \frac{TP + TN}{TP + TN + FP + FN} \quad 0 \leq AC \leq 1 \tag{7}$$

Das Ergebnis liegt dabei im Wertebereich zwischen 0 und 1. Werden die *Error Rate* und die *Accuracy* (oder auch *Non-Error Rate*) addiert, muss 1 beziehungsweise 100% herauskommen (Frank und Todeschini 1994).

Die nun folgenden Metriken messen die Leistung eines gelernten Modelles, im Gegensatz zur *Accuracy*, für eine bestimmte Klasse. Metriken, die sich auf eine bestimmte Klasse beziehen, werden laut (Sokolova, Japkowicz und Szpakowicz 2006) vor allem für die Klassifikation von Texten, die Extraktion von Informationen (beispielsweise APK-Features) und in der Bioinformatik benutzt, da in diesen Bereichen die Anzahl von Samples, die zu einer bestimmten (interessanten) Klasse gehören, meist wesentlich geringer ist als die Anzahl von vorhandenen Samples.

Precision

Die *Precision* (siehe Gleichung 8), auch *positiver Vorhersage-Wert* genannt, für eine bestimmte Klasse c gibt laut (Sokolova, Japkowicz und Szpakowicz 2006) den Anteil der richtig positiv erkannten Samples im Verhältnis zur Anzahl aller für c als positiv erkannten Samples an.

$$PR_c = \frac{TP_c}{TP_c + FP_c} \quad 0 \leq PR_c \leq 1 \tag{8}$$

Eine *Precision* von 1 besagt dementsprechend, dass nur Samples, die der Klasse c angehören, auch als Klasse c erkannt wurden.

Sensitivity

Die *Sensitivity*, auch *Recall* genannt, für eine Klasse c (siehe Gleichung 9) beschreibt laut (Frank und Todeschini 1994) den Anteil der korrekt als Klasse c klassifizierten Samples an der Gesamtheit aller Samples, die wirklich der Klasse c angehören (TP und FN).

$$SN_c = \frac{TP_c}{TP_c + FN_c} \quad 0 \leq SN_c \leq 1 \tag{9}$$

Entspricht der *Accuracy* für die Klasse c. Die *Sensitivity* ist 1, wenn alle Samples mit Klasse c auch als c klassifiziert wurden.

Specificity

Die *Specificity*, auch *Richtig-Negativ-Rate* genannt (siehe Gleichung 10), für eine Klasse c beschreibt laut (Frank und Todeschini 1994) den Anteil der korrekt nicht als Klasse c klassifizierten Samples an der Gesamtheit alle Samples, die nicht der Klasse c entsprechen.

$$SP_c = \frac{TN_c}{FP_c + TN_c} \quad 0 \leq SP_c \leq 1 \tag{10}$$

5.4 Verwendete Datensätze

Um die Forschungsfrage bestmöglich beantworten zu können, wurden mit Hilfe des implementierten „Feature Miners“ verschiedene Arten von Datensätzen für die weitere Analyse in *MatLab* generiert. Welche Verarbeitungs-Schritte zur Erstellung der Feature-Vektoren aus den extrahierten Rohdaten nötig waren und wie die Datensätze im Detail aussehen, kann den Kapiteln 4.2 beziehungsweise 4.3 entnommen werden.

Die Datensätze unterscheiden sich in folgenden zwei Bereichen:

1. Verwendeter Filter-Wert beim Erstellen der Datensätze (siehe Kapitel 4.1).
2. Welche Features in den Vektoren enthalten sind:
   - **F(ull):** Vollständige Vektoren
   - **S(tatic):** Nur Vektoren aus Features der statischen Analyse
   - **D(ynamic):** Nur Vektoren aus Features der dynamischen Analyse
   - **T(raffic):** Nur Vektoren aus Features der Netzwerkverkehrs-Analyse
   - **D(ynamic)+T(raffic):** Nur Vektoren aus Features der dynamischen Analyse von System-Aufrufen und Datei-Zugriffen sowie der Netzwerkverkehrs-Analyse

Aus den verwendeten APK-Datensätzen, die Schadsoftware sowie legitime Applikationen enthielten (siehe Kapitel 5.1), konnten durch den implementierten „Feature Miner“ insgesamt `123548` Features extrahiert werden. Diese teilen sich auf `63619` statische Features, `48395` Features aus der dynamischen Analyse von System-Aufrufen und Datei-Zugriffen und `11606` Features, die aus dem Netzwerkverkehr während der dynamischen Analyse extrahiert wurden, auf.

Label	Filter	Art	Feature-Anzahl			Features in %		
			S	D	T	S	D	T
50S	50	S	7538	0	0	11,84	0	0
50D	50	D	0	344	0	0	0,71	0
50T	50	T	0	0	137	0	0	1,18
50F	50	F	7538	344	137	11,84	0,71	1,18
30F	30	F	9130	436	193	14,35	0,90	1,66
F5P	5%	F	3297	2542	619	5,18	5,25	5,33
DT5P	5%	D+T	0	2542	619	0	5,25	5,33

Tabelle 11: Erstellte Datensätze mit Feature-Vektoren

Tabelle 11 zeigt eine Auflistung der Datensätze, die für eine Analyse mittels *MatLab* und der *Kohonen und CPANN Toolbox* erstellt wurden. Die in dieser Tabelle vergebenen Labels für bestimmte Datensätze werden im weiteren Verlauf der Arbeit zur Identifizierung dieser verwendet. Der Filter-Wert gibt an, bei wie vielen legitimen oder schädlichen Applikationen ein Feature gefunden werden muss, um dieses Feature in den Datensatz aufzunehmen. Die Feature-Anzahl, aufgeteilt in (S)tatic, (D)ynamic und (T)raffic, gibt an, wie viele Features der entsprechenden Analyse-Art ein Datensatz enthält. Die Prozent-Werte in den letzten drei Spalten geben jeweils an, wie hoch der prozentuale Anteil der extrahierten Features (Feature-Anzahl) an der Anzahl an Features ist, die gesamt bei einer der drei Analyse-Arten extrahiert wurden.

Besonders sind hierbei der Datensatz „F5P“, welcher vollständige Vektoren enthält, die aus rund 5% jeder Feature-Art bestehen (dafür verwendete Filter-Werte: `S=325`, `D=4` und `T=8`), sowie der Datensatz „DT5P“ hervorzuheben, welcher rund 5% aller dynamischen und Netzwerkverkehrs-Features enthält.

Die genauen Messwerte für jeden Datensatz können dem Anhang entnommen werden. Eine Zusammenfassung der Messergebnisse und eine Auflistung der genauen Messwerte für die Top-32 beziehungsweise Top-10 CP-ANN-Modelle erfolgt im folgenden Unterkapitel.

5.5 Messergebnisse

In diesem Unterkapitel wird auf die mittels *MatLab* und der *Kohonen und CPANN Toolbox* generierten CP-ANN-Modelle und deren Leistung näher eingegangen. Dazu wird zuerst die benötigte Zeit zum Errechnen eines bestimmten Modelles dessen gemessener *Error Rate* gegenübergestellt. Anschließend wird die *Error Rate* unter Verwendung verschiedener Datensätze, die entweder Vektoren aus statischer, dynamischer oder einer Netzwerkverkehrs-Analyse enthalten, mit einem Datensatz, der vollständige Feature-Vektoren enthält, verglichen. Danach wird untersucht, welchen Einfluss verschiedene Filter-Werte im „Feature Miner“ bei der Generierung von Datensätzen haben. Abschließend werden die am besten für die Klassifikation in legitime und schädliche Applikationen geeigneten CP-ANN-Modelle präsentiert.

5.5.1 Error Rate im Verhältnis zur benötigten Zeit

Abbildung 28 zeigt die *Error Rate* verschiedener CP-ANN-Modelle im Vergleich mit der benötigten Zeit, um diese Modelle zu berechnen. Hierfür wurde der „F50“-Datensatz verwendet. Da sich die Anzahl von Neuronen, die ein CP-ANN enthält, quadratisch zur *Grid-Size* (n_G) verhält, steigt auch die benötigte Zeit zum Berechnen eines Modells mit der Erhöhung der *Grid-Size* quadratisch an ($O(n^2)$). Wie viele Epochen (n_E) ein Modell trainiert wird, verändert die Laufzeit zusätzlich linear ($O(n)$). Die Zeit zum Berechnen eines bestimmten Modelles beträgt dementsprechend $O(n_E \cdot n_G^2)$. Die *Error Rate* geht hingegen mit steigender *Grid-Size* immer weiter Richtung 0, was einer perfekten Trennung von schädlichen und legitimen Applikationen entsprechen würde. Ab einer *Grid-Size* von

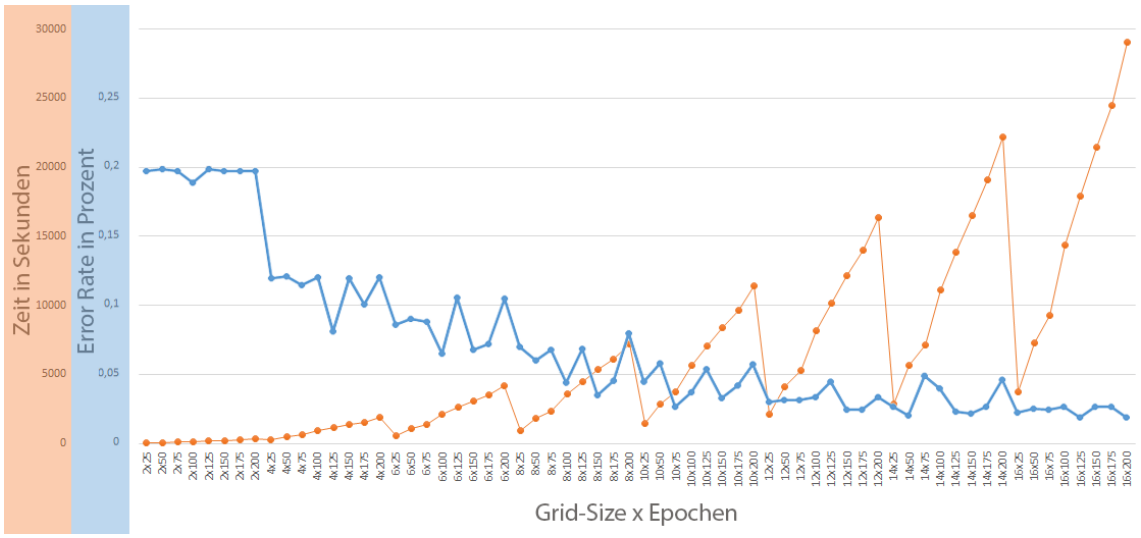


Abbildung 28: Error Rate im Verhältnis zur benötigten Laufzeit

12 konnten, mit Ausnahme von 14×75 und 14×100, auch mit wenigen Epochen gute Ergebnisse erzielt werden.

5.5.2 Einfluss verschiedener Arten von Feature-Vektoren

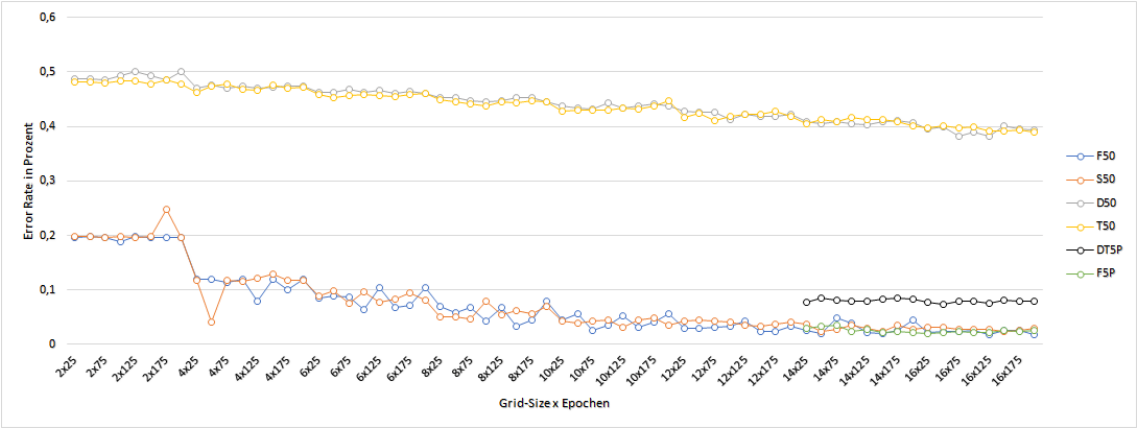
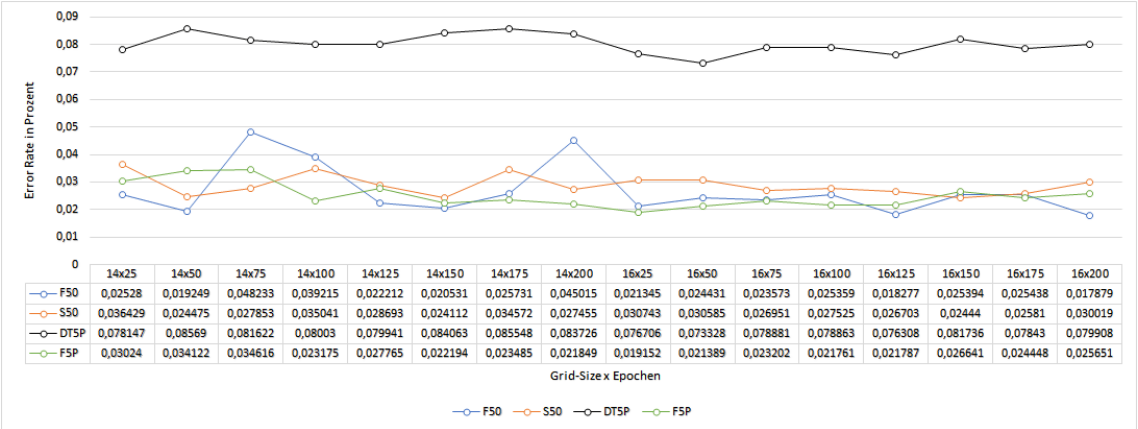


Abbildung 29: Einfluss verschiedener Arten von Feature-Vektoren (Grid-Size 2-16)

Abbildung 29 zeigt, dass sowohl Datensätze mit Feature-Vektoren aus einer dynamischen Analyse von System-Aufrufen und Datei-Zugriffen alleine (gelb markiert), als auch aus der Analyse des Netzwerk-Verkehres während der dynamischen Analyse alleine (grau markiert) nur bedingt geeignet sind, um legitime und schädliche Applikationen richtig zu klassifizieren. Die *Error Rate* beträgt in beiden Fällen unter allen getesteten Modellen zwischen rund 40% und 50%. Grund hierfür könnte allerdings sein, dass die Feature-Vektoren dieser beiden Datensätze über eine erheblich geringere Anzahl an Gewichten

verfügen (siehe Tabelle 11). Der Datensatz „DT5P“ (schwarz markiert), welcher rund 5% aller Features aus der dynamischen Analyse (System-Aufrufe, Datei-Zugriffe sowie Netzwerkverkehr) enthält, schnitt wesentlich besser ab. Die *Error-Rate* bewegte sich hier zwischen rund 0,085% und 0,073%. Daten aus der statischen Analyse alleine (orange markiert) liefern noch bessere Ergebnisse. Ab einer *Grid-Size* von 6 beläuft sich der prozentuale Anteil falsch klassifizierter Samples auf unter 10% und nähert sich mit steigender *Grid-Size* immer weiter einer perfekten Klassifikation an. Nur mit den blau sowie mit den grün markierten vollständigen Datensätzen, die verschiedene Anteile von allen drei Feature-Arten enthalten (statische, dynamische und Netzwerkverkehrs-Features), kann dieses Ergebnis noch einmal geringfügig verbessert werden, was allerdings in Abbildung 30 besser zu erkennen ist.



	14x25	14x50	14x75	14x100	14x125	14x150	14x175	14x200	16x25	16x50	16x75	16x100	16x125	16x150	16x175	16x200
F50	0,02528	0,019249	0,048233	0,039215	0,022212	0,020531	0,025731	0,045015	0,021345	0,024431	0,023573	0,025359	0,018277	0,025394	0,025438	0,017879
S50	0,036429	0,024475	0,027853	0,035041	0,028693	0,024112	0,034572	0,027455	0,030743	0,030585	0,026951	0,027525	0,026703	0,02444	0,02581	0,030019
DT5P	0,078147	0,08569	0,081622	0,08003	0,079941	0,084063	0,085548	0,083726	0,076706	0,073328	0,078881	0,078863	0,076308	0,081736	0,07843	0,079908
F5P	0,03024	0,034122	0,034616	0,023175	0,027765	0,022194	0,023485	0,021849	0,019152	0,021389	0,023202	0,021761	0,021787	0,026641	0,024448	0,025651

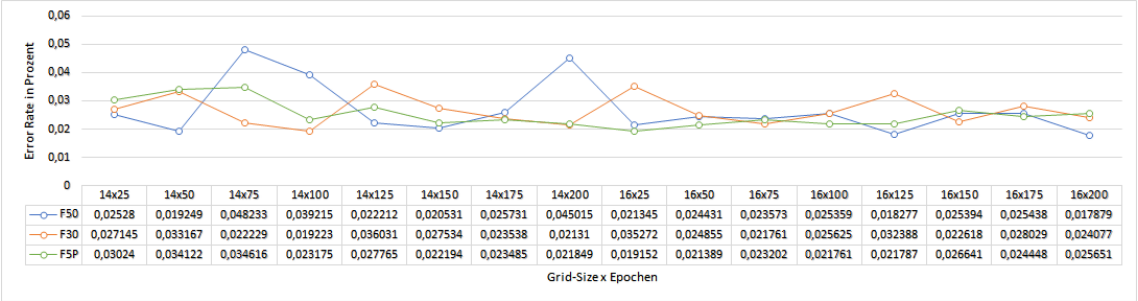
Abbildung 30: Einfluss verschiedener Arten von Feature-Vektoren (Grid-Size 14-16)

In Abbildung 30 sind die berechneten *Error Rates* noch einmal genauer zu sehen. Der Plot wurde dafür auf die beiden *Grid-Sizes* 14 und 16 eingeschränkt. Auch die beiden Datensätze „D50“ und „T50“ wurden für eine bessere Darstellung entfernt. Zu sehen ist, dass vor allem der „F5P“ Datensatz gute Ergebnisse liefert.

5.5.3 Einfluss der Filter-Werte beim Erstellen der Datensätze

Für die Analyse von Datensätzen, die auf verschiedenen Filter-Werten bei deren Erstellung durch den implementierten „Feature Miner“ beruhen, wurden nur noch 14 beziehungsweise 16 Epochen untersucht, da damit auch in den vorangegangenen Versuchen gute Ergebnisse erzielt werden konnten. Das Training wurde allerdings, wie bereits zuvor, über $[25, 50, 75, 100, 125, 150, 175, 200]$ Epochen durchgeführt.

Abbildung 31 zeigt, wie sich verschiedene Filter-Werte bei der Erstellung von Datensätzen mittels des entwickelten „Feature Miners“ auswirken. Die getesteten Datensätze enthielten jeweils vollständige Feature-Vektoren aus allen drei Kategorien. Features, die in den Vektoren enthalten sind, mussten dafür in mindestens 50 beziehungsweise 30 schädlichen oder legitimen Applikationen vorkommen. Der zusätzliche Datensatz „F5P“ stellt dabei



	14x25	14x50	14x75	14x100	14x125	14x150	14x175	14x200	16x25	16x50	16x75	16x100	16x125	16x150	16x175	16x200
F50	0,02528	0,019249	0,048233	0,039215	0,022212	0,020531	0,025731	0,045015	0,021345	0,024431	0,023573	0,025359	0,018277	0,025394	0,025438	0,017879
F30	0,027145	0,033167	0,022229	0,019223	0,036031	0,027534	0,023538	0,02131	0,035272	0,024855	0,021761	0,025625	0,032388	0,022618	0,028029	0,024077
F5P	0,03024	0,034122	0,034616	0,023175	0,027765	0,022194	0,023485	0,021849	0,019152	0,021389	0,023202	0,021761	0,021787	0,026641	0,024448	0,025651

Abbildung 31: Verschiedene Filter-Werte (F50, F30, F5P) im Vergleich

einen Spezialfall dar. Dieser enthält jeweils die rund 5% der, auf jede der drei Analyse-Arten bezogen, am häufigsten vorkommenden Features. Zu sehen ist, dass die Ergebnisse ab einer *Grid-Size* von 16 nur mehr gering variieren. Der „F50“ Datensatz, der insgesamt 8019 verschiedene Features enthält, schneidet bis auf drei Ausreißer bei einer *Grid-Size* von 14 und 75, 100 beziehungsweise 200 Epochen relativ gut ab. Der „F30“ Datensatz, der 9759 Features enthält und damit um 1740 Features mehr als der „F50“ Datensatz, schneidet vor allem bei den zuvor genannten Ausreißern des „F50“ Datensatzes gut ab. Der Datensatz „F5P“, welcher insgesamt 6458 Features enthält, liefert hier die konstantesten Ergebnisse. Die gemessenen *Error Rates* befanden sich dabei oft zwischen denen der beiden anderen Datensätze beziehungsweise meist im unteren Messbereich.

5.5.4 Top-32 beziehungsweise Top-10 berechnete Modelle

Tabelle 23 im Anhang zeigt eine Auflistung der 32 beziehungsweise 10 CP-ANN-Modelle, die in dieser Arbeit am besten abgeschnitten haben, zusammen mit deren Messwerten. Die *Specificity*, *Sensitivity* und *Precision* wurden dabei für beide Klassen, legitime Applikationen und Schadsoftware, einzeln berechnet.

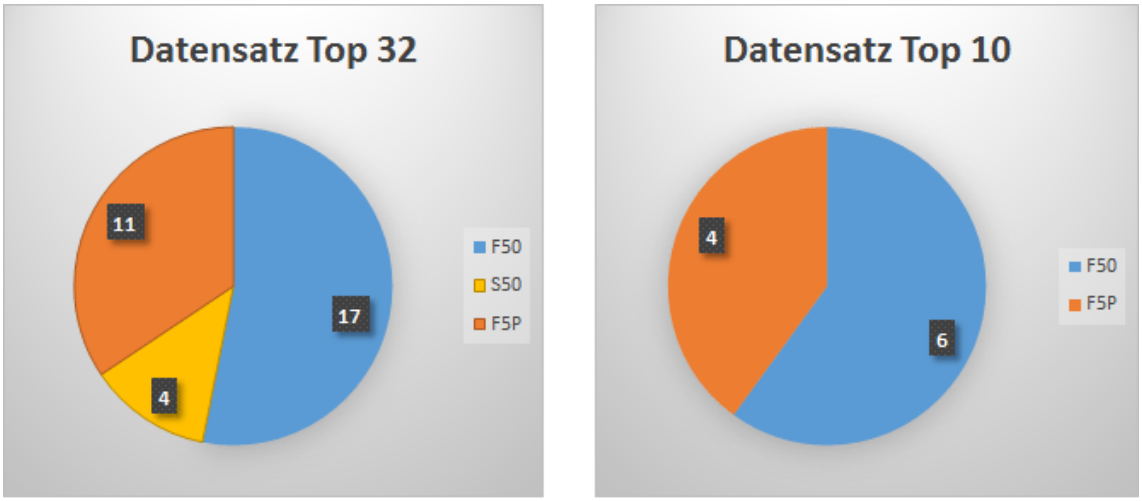


Abbildung 32: Auswertung: Top-32 und Top-10 Datensätze

Abbildung 32 zeigt, welche Datensätze für die Top-32 beziehungsweise die Top-10 Modelle Verwendung fanden. Unter den Top-32 Modellen wurden mehr als die Hälfte (17 Modelle) auf Basis des „F50“ Datensatzes berechnet. Vier Modelle, wenn diese sich auch in der unteren Hälfte der Tabelle 23 befinden, wurden sogar alleine mit Vektoren aus der statischen Analyse berechnet, was zeigt, dass eine alleinige statische Analyse bereits sehr gute Ergebnisse bei gleichzeitig geringem Aufwand, um die entsprechenden Features zu extrahieren, liefern kann. Der Datensatz „F5P“, welcher rund 5% aller Features enthält, die von der jeweiligen Feature-Art extrahiert werden konnten, kam als Grundlage für 11 der Top-32 CP-ANN-Modelle zum Einsatz. Unter den Top-10 waren die beiden grundverschiedenen Datensätze „F50“ und „F5P“ mit 6 beziehungsweise 4 Vorkommen vertreten. Es sollte jedoch erwähnt werden, dass der Datensatz „F5P“ unter den Top 5 nur ein Mal (an dritter Stelle) vertreten war, jedoch für die Berechnung des dazugehörigen Modells nur 25 Epochen benötigte und somit nur rund 1/10 der Zeit des siegenden CP-ANN-Modelles.

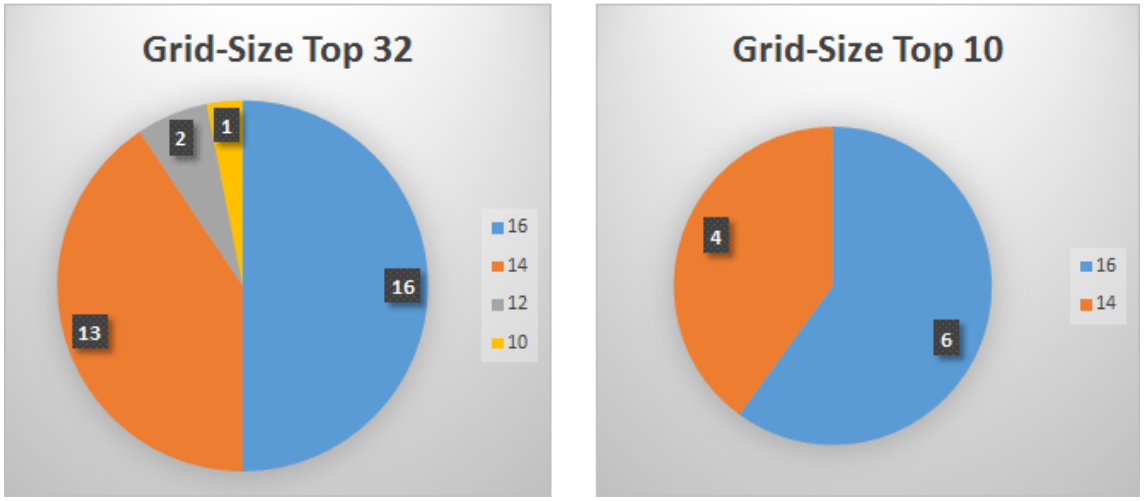


Abbildung 33: Auswertung: Top-32 und Top-10 Grid-Sizes

Abbildung 33 zeigt in einer Übersicht, wie hoch die *Grid-Sizes* zur Berechnung der Top-32 beziehungsweise Top-10 CP-ANN-Modelle waren. Bei 16 der Top-32 Modelle lieferte eine *Grid-Size* von 16 die besten Ergebnisse, gefolgt von einer *Grid-Size* von 14, die bei 13 der Top-32 CP-ANN-Modellen als Grundlage diente. Für immerhin zwei CP-ANN-Modelle unter den Top-32 war eine *Grid-Size* von 12 ausreichend, während ein Modell sogar mit einer *Grid-Size* von 10 eine gute *Accuracy* lieferte. Unter den Top-10 diente für 60% der berechneten Modelle eine *Grid-Size* von 16 als Grundlage, während sich die restlichen 40% auf eine *Grid-Size* von 14 verteilten. Es kann durchaus gesagt werden (siehe auch die vollständigen Ergebnistabellen im Anhang), dass bei den errechneten Modellen ein maßgeblicher Zusammenhang zwischen der *Grid-Size* und einer guten *Accuracy* besteht.

Abbildung 34 zeigt zusammengefasst, wie viele Epochen für das Training der Top-32 beziehungsweise Top-10 CP-ANN-Modelle benötigt wurden. Nur zwei der Top-32 Modelle kamen mit nur 25 Epochen für das Training aus. Interessant dabei ist jedoch, dass sich diese zwei Modelle auch unter den Top-10 befinden. Mit 50 Lern-Epochen wurden 5 der

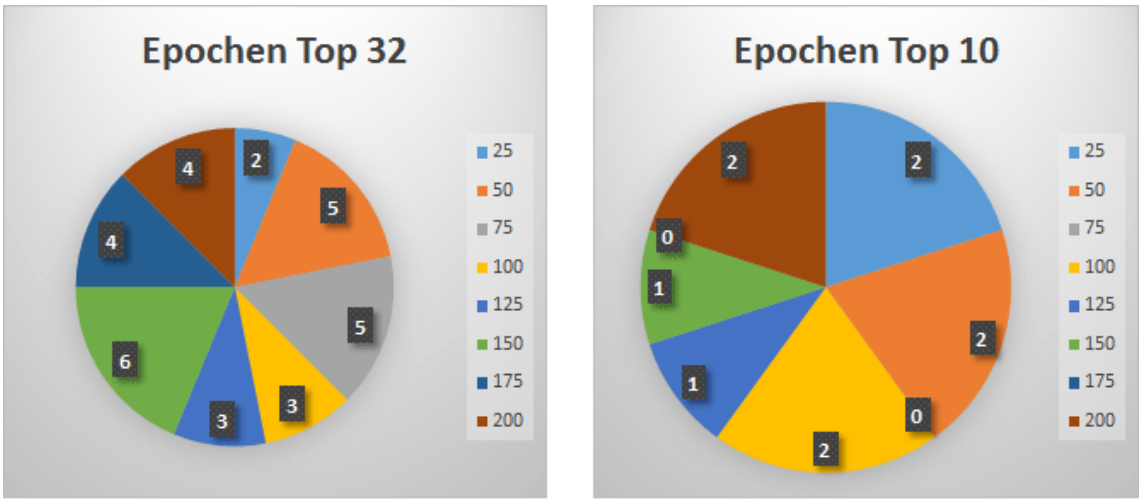


Abbildung 34: Auswertung: Top-32 und Top-10 Epochen

Top-32 Modelle trainiert, zwei davon finden sich außerdem unter den Top-10. Für fünf der Top-32 CP-ANN-Modelle waren 75 Epochen nötig, um diese zu trainieren. Keine Modelle schafften es jedoch mit nur 75 Lern-Epochen in die Top-10. Drei der Top-32 Modelle wurden mit 100 Epochen trainiert, davon schafften es immerhin zwei auch in die Top-10. Ebenfalls drei der Top-32 Modelle benötigten 125 Epochen, nur eines dieser Modelle schaffte es in die Top-10. Die meisten Modelle unter den Top-32, nämlich 6, benötigten 150 Lern-Epochen, allerdings befindet sich nur eines dieser Modelle unter den Top-10. Vier der Top-32 CP-ANN Modelle wurden 175 Epochen lang trainiert. Keines dieser Modelle schaffte es in die Top-10. Weitere vier Modelle benötigten 200 Epochen für ihr Training. Nur zwei davon schafften es in die Top-10, darunter allerdings auch das CP-ANN Modell, das die beste *Accuracy* unter allen errechneten Modellen erreichte. Interessant ist, dass unter den Top-32 CP-ANN-Modellen zwar 17 und damit knapp über die Hälfte mehr als 100 Epochen benötigten, allerdings 60% der unter den Top-10 vertretenen Modellen mit maximal 100 Epochen auskamen. Vier der Top-10 benötigten dafür sogar nur maximal 50 Trainings-Epochen. Da sich die Zeit, die aufgewendet werden muss, um ein CP-ANN-Modell zu berechnen (siehe Abbildung 28), auch maßgeblich an der Anzahl der Trainings-Epochen orientiert, erscheinen besonders diese Modelle interessant.

Die beste *Accuracy* (siehe Gleichung 7) lieferte ein Modell, das den „F50“-Datensatz als Grundlage hatte, über eine *Grid-Size* von 16 verfügte und 200 Epochen lang trainiert wurde. Dieses Modell erreichte eine *Accuracy* von $0,982$, brauchte dafür allerdings $29055,2$ Sekunden. An dritter Stelle befindet sich ein Modell (*Grid-Size* 16 und 25 Lern-Epochen), das mit dem „F5P“-Datensatz als Grundlage in $2946,9$ Sekunden (rund 1/10 der Zeit des Sieger-Modelles) eine *Accuracy* von $0,981$ erreichte. Das Modell an der 10. Stelle konnte noch eine *Accuracy* von $0,978$ erreichen, während das Modell an Stelle 32 eine *Accuracy* von $0,975$ erreichte. Zu sehen ist, dass die trainierten CP-ANN-Modelle allesamt sehr nahe beieinander liegen, was deren *Accuracy* betrifft.

Bei der gemessenen *Specificity* beziehungsweise der *Richtig-Negativ-Rate* (siehe Gleichung

10) ergab sich ein Durchschnittswert von 0,970 für die Top-32, beziehungsweise 0,972 für die Top-10 legitimen Applikationen. Die Schadsoftware-Klasse erreichte eine leicht höhere durchschnittliche *Specificity* von 0,984 für die Top-32, beziehungsweise 0,987 für die Top-10 CP-ANN-Modelle.

Bei der gemessenen *Sensitivity*, beziehungsweise dem *Recall* einer bestimmten Klasse, wurden für legitime Applikationen und die Top-32 Modelle ein Durchschnittswert von 0,984, beziehungsweise für die Top-10 Modelle ein Durchschnittswert von 0,987 erreicht. Die Schadsoftware-Klasse erreichte eine leicht niedrigere durchschnittliche *Sensitivity* von 0,970 für die Top-32, beziehungsweise 0,972 für die Top-10 CP-ANN-Modelle.

Für die Top-32 CP-ANN-Modelle und die Klasse der legitimen Applikationen konnte eine durchschnittliche *Precision* (siehe Gleichung 8) von 0,970, beziehungsweise für die Top-10 eine durchschnittliche *Precision* von 0,972 erreicht werden. Die Schadsoftware-Klasse schnitt unter den Top-32 mit 0,985, beziehungsweise mit 0,987 für die Top-10 Modelle leicht besser ab.

Damit kann zusammenfassend gesagt werden, dass sowohl die Top-32 als auch die Top-10 CP-ANN-Modelle geringfügig besser in der Lage waren Schadsoftware-Samples zu klassifizieren als legitime Applikationen.

5.5.5 Vorhersage unbekannter Samples

Da in der Realität oftmals nicht für jede Schadsoftware-Familie oder -Variante Samples zum Trainieren vorhanden sind, weil diese beispielsweise bisher unentdeckt blieben, sollte ein Modell auch in der Lage sein, zuvor unbekannte Samples richtig zu klassifizieren. Um dies zu testen, wurde auf die Top-10 der oben vorgestellten CP-ANN Modelle eine vierfache Kreuz-Validierung angewandt.

Bei der k-fachen Kreuz-Validierung wird der Datensatz D in k möglichst gleich große Teil-Datensätze (*Folds*) $(D_1, D_2, ..., D_k)$ aufgeteilt, deren enthaltene Samples sich nicht überschneiden. Das Modell wird dabei k Mal mit $t \in \{1, 2, ..., k\}$ trainiert. Dabei wird für das Training der Modelle jeweils $D \backslash D_t$ und für die Überprüfung des trainierten Modells D_t verwendet (Kohavi u. a. 1995).

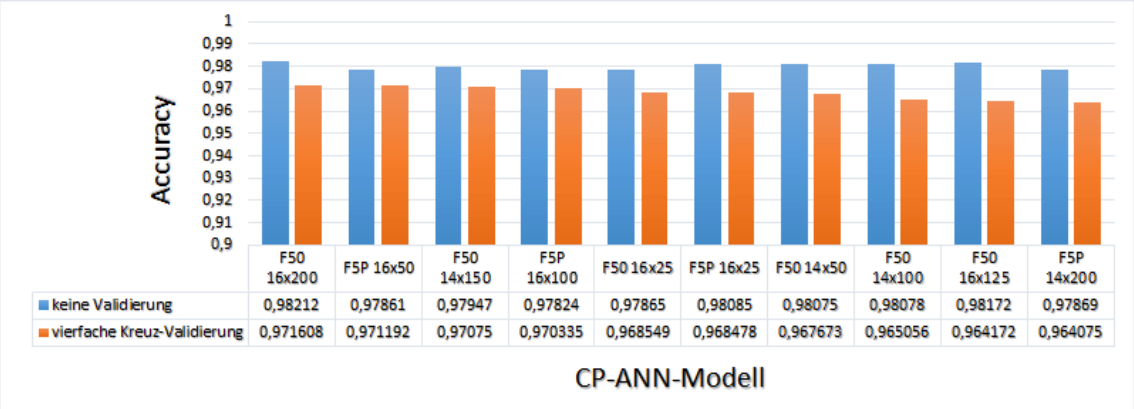
Die kreuz-validierte *Accuracy* eines Modells ergibt sich anschließend aus dem Durchschnitt der k berechneten *Accuracy*-Werte. Auch für alle anderen Metriken kann so ein kreuzvalidiertes Ergebnis berechnet werden.

Die *Kohonen und CPANN Toolbox* unterstützt zwei Kreuz-Validierungs-Verfahren für die Überprüfung von CP-ANNs:

1. **Venetian Blinds:** Bei diesem Verfahren wird der Datensatz, am Beispiel einer dreifachen Kreuz-Validierung, folgendermaßen aufgeteilt: Der erste Teil-Datensatz D_1 enthält $[t, 0, 0, t, 0, 0, ..., t, 0, 0]$, der zweite Datensatz D_2 enthält $[0, t, 0, 0, t, 0, ..., 0, t, 0]$ und der dritte Datensatz D_3 entsprechend $[0, 0, t, 0, 0, t, ..., 0, 0, t]$.

2. **Contiguous Blocks:** Bei diesem Verfahren wird der Datensatz D einfach in k Teil-Datensätze zerteilt. Bei einem Datensatz mit 12 Samples und einer dreifachen Kreuz-Validierung würde der erste Teil-Datensatz $[t, t, t, t, 0, 0, 0, 0, 0, 0, 0, 0]$ enthalten, der zweite $[0, 0, 0, 0, t, t, t, t, 0, 0, 0, 0]$ und der dritte $[0, 0, 0, 0, 0, 0, 0, 0, t, t, t, t]$.

Für diese Arbeit wurde das *Venetian Blinds* Verfahren gewählt, da der APK-Datensatz für viele Schadsoftware-Familien nur wenige Samples enthält und durch das *Contiguous Blocks* Verfahren bei einer vierfachen Kreuz-Validierung bei zwei von vier Durchläufen rund die Hälfte aller Schadsoftware-Samples nicht für das Training verwendet würden.



	F50 16x200	F5P 16x50	F50 14x150	F5P 16x100	F50 16x25	F5P 16x25	F50 14x50	F50 14x100	F50 16x125	F5P 14x200
keine Validierung	0,98212	0,97861	0,97947	0,97824	0,97865	0,98085	0,98075	0,98078	0,98172	0,97869
vierfache Kreuz-Validierung	0,971608	0,971192	0,97075	0,970335	0,968549	0,968478	0,967673	0,965056	0,964172	0,964075

Abbildung 35: Vorhersage unbekannter Samples

Abbildung 35 zeigt eine Gegenüberstellung der bei den Top-10 CP-ANN-Modellen gemessenen *Accuracy*-Werte, jeweils ohne und mit vierfacher Kreuz-Validierung. Zu sehen ist, dass die Modelle bei einer Kreuz-Validierung zwar allesamt schlechter abschneiden, der durchschnittliche Unterschied hierbei liegt allerdings nur bei $1,18\%$. Besonders hervorzuheben sind hierbei die Modelle „F5P 16x50“ mit einer *Accuracy* von rund $0,971$, „F50 16x25“ mit einer *Accuracy* von rund $0,969$, „F5P 16x25“ mit einer *Accuracy* von rund $0,968$ und „F50 14x50“ mit einer *Accuracy* von rund $0,968$, da diese relativ wenig Zeit für deren Berechnung benötigten.

Dass der durchschnittliche Unterschied zwischen den errechneten CP-ANN-Modellen mit beziehungsweise ohne vierfache Kreuz-Validierung nur bei rund $1,18\%$ liegt, zeigt, dass sich CP-ANNs auch für die akkurate Bestimmung von zuvor unbekannten Samples gut eignen.

Schlechtere *Accuracy* bei Kreuz-Validierung

Es gibt laut (Tetko, Livingstone und Luik 1995) zwei Gründe, warum sich ein gelerntes Modell, das eine hervorragende Leistung, beispielsweise eine sehr gute *Accuracy* liefert, unter Umständen nicht genauso gut für die Vorhersage von unbekannten Samples eignen kann. Ein ANN, das über eine ausreichende Anzahl von Neuronen verfügt, ist in der Lage, ein bestimmtes Trainings-Set exakt zu lernen. Dabei ist es allerdings auch möglich, dass

dieses Trainings-Set zu exakt gelernt wird, wodurch die Generalisierungsfähigkeit, also die Fähigkeit, auch unbekannte *Muster* zu erkennen (siehe Kapitel 3.4), des ANN leidet. Die beiden Voraussetzungen, die eine schlechte Generalisierung zur Folge haben können, sind:

1. Die Größe (*Grid-Size*) des ANNs (Overfitting)
2. Die Anzahl der Trainings-Epochen (Overtraining)

Overfitting

Das *Overfitting*-Problem beschreibt bei ANNs das Überschreiten der optimalen Anzahl an vorhandenen Neuronen. Laut (Tetko, Livingstone und Luik 1995) hat *Overfitting* im Falle von ANNs allerdings keinen Einfluss auf die Generalisierungsfähigkeit, solange *Overtraining* verhindert wird.

Overtraining

Das weitaus größere Problem stellt laut (Tetko, Livingstone und Luik 1995) das sogenannte *Overtraining* dar. Darunter wird das Überschreiten einer bestimmten Anzahl von Trainings-Epochen verstanden, nach denen das zu Grunde liegende Trainings-Set von einem ANN so exakt gelernt wird, dass dieses nicht mehr in der Lage ist zu generalisieren und somit auch nicht mehr in der Lage ist, zuvor unbekannte Samples richtig zu klassifizieren. Der Grund dafür liegt in der Funktionsweise von ANNs. Diese lernen anfangs immer die grobe Struktur der Daten im Trainings-Set. In späteren Trainings-Epochen wird die gelernte Struktur immer weiter verfeinert, bis schließlich ab einem bestimmten Punkt auch Strukturen, die nur einem Rauschen im Datensatz entsprechen, gelernt werden. Wann der am besten geeignete Punkt ist, um das Netzwerk-Training zu beenden, kann laut (Tetko, Livingstone und Luik 1995) durch Kreuz-Validierung mit verschiedenen Datensätzen herausgefunden werden.

6 Diskussion & Ausblick

Im ersten Teil dieses Kapitels wird auf die Beiträge eingegangen, die durch diese Arbeit für das Forschungsgebiet geleistet wurden. Anschließend werden die gemessenen Ergebnisse kritisch diskutiert und die Forschungsfrage beantwortet. Im zweiten und abschließenden Teil wird ein Ausblick auf potentielle weitere Entwicklungen gegeben.

6.1 Diskussion

In diesem Unterkapitel werden der erarbeitete Lösungsweg und die resultierenden Ergebnisse diskutiert. Außerdem wird auf die für das Forschungsgebiet geleisteten Beiträge eingegangen und die Forschungsfrage beantwortet.

Durch die im Zuge dieser Master Thesis getätigte Arbeit konnten für das Forschungsgebiet, welches sich mit der automatisierten Identifikation von Android-Schadsoftware auseinander setzt, vor allem zwei Beiträge geleistet werden.

Einerseits wurde mit dem „AndroSOM Feature Miner“ ein effizientes Framework zum Extrahieren von Featues aus Android-Applikationen entwickelt. Dieses ist nicht nur in der Lage, mit Hilfe von *Androguard* eine statische Analyse von APK-Dateien durchzuführen, sondern ermöglicht es auch, eine Applikation in einer standardisierten Umgebung, der „AndroSOM“-Sandbox, auszuführen und Daten über System-Aufrufe, Datei-Zugriffe und Netzwerk-Aktivitäten zu sammeln. Während der Laufzeit der Applikation in der Sandbox werden Daten aus der vorangegangenen statischen Analyse genutzt, um auf die zu testende Applikation zugeschnittene Events zu generieren, die gegebenenfalls schädliches Verhalten auslösen. Beim „AndroSOM Feature Miner“ handelt es sich dementsprechend um ein Werkzeug, das statische und dynamische Analyse kombiniert, um bestmöglich Daten über eine bestimmte Applikation zu sammeln. Ähnliche Frameworks existieren zwar, beispielsweise die *AASandbox* von (Blasing u. a. 2010), *Andrubis* von (Lindorfer 2012), *AppsPlayground* von (Rastogi, Chen und Enck 2013) oder die *Mobile-Sandbox* von (Spreitzenbarth u. a. 2013), verwenden allerdings allesamt den Standard-Android-Emulator *QEMU*, der im Android SDK enthalten ist. Dieser emuliert in seiner Standard-Konfiguration die ARM-Architektur, wodurch die Reaktionszeiten des Android-Systems in der Sandbox und die der darin ausgeführten Applikationen sehr schlecht sind. In dieser Arbeit wird stattdessen auf *VirtualBox* als VMM gesetzt, wodurch ein für die „AndroSOM“-Sandbox kompiliertes Android-x86 4.4 System genutzt werden kann. Dadurch, dass keine ARM-Architektur auf dem Host-System emuliert werden muss, verbessert sich die Reaktionszeit des System so stark, dass eine Analyse mittels `strace` des *Zygote*-Prozesses und allen Prozessen, die von diesem Prozess *geforkt* werden, ermöglicht wird. Beim *Zygote*-Prozess handelt es sich um die erste Instanz der Dalvik-VM. Alle Prozesse, die zur unter Beobachtung stehenden Applikation gehören, können somit von der dynamischen Analyse abgedeckt werden. Aus den gesammelten Daten der statischen und der dynamischen Analyse können in weiterer Folge verschiedene Feature-Vektoren für die anschließende Analyse in *MatLab* generiert werden.

Andererseits konnte bewiesen werden, dass sich CP-ANN durchaus für die Identifizierung von (unbekannten) Schadsoftware-Samples eignen. Bei CP-ANN handelt es sich um eine Weiterentwicklung von SOM, welche diese um die Möglichkeit erweitert, überwachtes Lernen durchzuführen. SOM wurden bereits von (Barrera u. a. 2010) erfolgreich zur Analyse von Applikations-Berechtigungen, einem der grundlegenden Sicherheitsmechanismen des Android-Systems, eingesetzt. Diese Arbeit unterscheidet sich dahingehend, dass nicht Berechtigungen alleine als Grundlage für die SOM beziehungsweise CP-ANN herangezogen werden, sondern Features, die aus statischer und dynamischer Analyse stammen und mittels des entwickelten Frameworks aus 2185 Applikationen, darunter 1080 Schadsoftware-Samples und 1105 legitime Applikationen aus dem *Google Play Store*, extrahiert wurden.

Resultate

Es konnte gezeigt werden, dass es mit Hilfe von CP-ANNs möglich ist, schädliche von legitimen Android Applikationen zu unterscheiden, auch wenn es sich um zuvor unbekannte Samples handelt (siehe Abbildung 35). Dabei wurden ähnlich gute, teils auch bessere Ergebnisse erzielt als in vorangegangenen, ähnlichen Arbeiten, beispielsweise von (Shabtai u. a. 2012) oder (Schmidt u. a. 2009) in denen allerdings nur Features aus einer statischen beziehungsweise dynamischen Analyse alleine als Grundlage dienten.

Dabei wurde festgestellt, dass nicht nur CP-ANN-Modelle gute Ergebnisse liefern, die durch eine hohe Anzahl an Trainings-Epochen verhältnismäßig viel Zeit für deren Berechnung beanspruchen, sondern, vor allem nach einer durchgeführten vierfachen Kreuz-Validierung, auch CP-ANN-Modelle, die nur bis zu 50 Epochen trainiert wurden, sehr gut abschnitten. Grund für das gute Abschneiden von Modellen mit nur wenigen Trainings-Epochen könnte das bekannte Problem des *Overtrainings* sein. Dieses besagt, dass ein ANN-Modell, welches über zu viele Epochen hinweg trainiert wird, seine Fähigkeit zu generalisieren verliert. Dadurch steigt zwar die *Accuracy*, wenn es sich um bekannte Samples handelt, nimmt allerdings ab, wenn zuvor unbekannte Samples klassifiziert werden sollen.

Auch bei den in dieser Arbeit identifizierten Top-10 CP-ANN-Modellen nahm die *Accuracy* nach einer durchgeführten vierfachen Kreuz-Validierung, die unbekannte Samples simulieren sollte, ab. Der durchschnittliche Unterschied hierbei lag allerdings nur bei vertretbaren $1,18\%$.

Es konnte außerdem beobachtet werden, dass sich eine *Grid-Size* ab 12 (144 Neuronen) sehr gut für die Klassifikation der Samples eignete. Eine obere Grenze für die *Grid-Size* konnte nicht gefunden werden, da aus Ressourcen-Gründen nur bis zu einer *Grid-Size* von 16 (256 Neuronen) getestet wurde.

Ebenfalls konnte gezeigt werden, dass es durchaus einen Unterschied macht, ob die Feature-Vektoren nur aus der statischen, dynamischen oder der Analyse des Netzwerkverkehrs während der dynamischen Analyse stammten. Feature-Vektoren aus der statischen Analyse schnitten hierbei am besten ab. Auch mit Feature-Vektoren aus der dynamischen Analyse alleine konnten bei einer ausreichend hohen Anzahl an Gewichten gute Ergebnisse erzielt werden. Hier besteht allerdings, vor allem bei den Feature-Vektoren aus der dynamischen Analyse, noch Potential für Verbesserungen (siehe Kapitel 6.2).

6.2 Ausblick

In diesem Unterkapitel wird ein Ausblick auf mögliche Weiterentwicklungen der in dieser Arbeit gewählten Techniken und Methoden gegeben. Dabei wird zum einen auf das entwickelte Framework und zum anderen auf Möglichkeiten eingegangen, noch besser geeignete CP-ANN-Modelle zum Identifizieren von zuvor unbekannten schädlichen Android Applikationen zu generieren.

Feature Miner

Um den implementierten „Feature Miner“ zu einem Produkt weiterentwickeln zu können, beispielsweise für den Einsatz in einem Applikations-Marktplatz, um schädlichen Applikationen zu erkennen, könnte das Framework dahingehend angepasst werden, dass es als Service in der Cloud lauffähig wäre. Außerdem müsste ein weiteres Modul, das die Logik zum Trainieren von CP-ANN-Modellen enthält und das Klassifizieren von Android Applikationen ermöglicht, entwickelt werden. Diese beiden Aufgaben werden momentan durch *MatLab*, beziehungsweise die *Kohonen und CPANN Toolbox* abgedeckt.

Statische Analyse

Um die statische Analyse zu verbessern, wäre es denkbar, Code von Drittanbietern, beispielsweise von Werbe- oder Analyse-Netzwerken, zu entfernen. Außerdem erscheint es sinnvoll, weitere Daten über Applikationen zu sammeln. Diese könnten vor allem dazu beitragen, die dynamische Analyse zu unterstützen.

Dynamische Analyse

Durch die aktuelle Architektur des Frameworks ist momentan keine Parallelisierung der dynamischen Analyse möglich. Dies wäre allerdings sinnvoll, um größere APK-Datensätze effizient zu analysieren. Ein weiterer Ansatzpunkt, um die dynamische Analyse, vor allem im Hinblick auf den Ressourcen-Verbrauch (CPU), effizienter zu gestalten, ist der Einsatz von Virtual Machine Introspection (VMI), wie sie beispielsweise bei *DroidScope* oder *CopperDroid* (siehe Kapitel 3.2.2) zum Einsatz kommt. Bei der VMI befindet sich der Analyse-Code außerhalb des ausgeführten Android-Systems, direkt im Emulator oder dem VMM. Damit wäre es möglich, `strace` gänzlich zu ersetzen.

Ein Umstieg auf VMI wäre auch in Hinblick auf Android 5.0 und die dort enthaltene neue Standard-Runtime ART, welche die Dalvik-VM ersetzt, sinnvoll. Momentan besteht eine starke Abhängigkeit zwischen der dynamischen Analyse von System-Aufrufen und Datei-Zugriffen mittels `strace` und dem *Zygote*-Prozess, einer Instanz der Dalvik-VM.

Um auch Applikationen zu untersuchen, die auf nativen ARM-Bibliotheken aufbauen und womöglich keine entsprechenden x86-Bibliotheken enthalten, könnte eine ARM-Sandbox als *Ausweichlösung* implementiert werden. Dies wäre allerdings aus Performance-Gründen nur mit einem Verzicht auf `strace` sinnvoll, was wiederum für VMI sprechen würde.

In der Initialisierungs-Phase eines dynamischen Analyse-Durchlaufes werden alle Broadcast-Receiver einer zu untersuchenden Applikation aus deren Manifest-Datei ausgelesen. Diese werden anschließend zur Stimulation der Applikation genutzt, um gegebenenfalls schädliches Verhalten auszulösen. Broadcast-Receiver können allerdings auch während der Laufzeit einer Applikation dynamisch registriert werden. In diesem Fall stehen diese dem Framework momentan nicht für die Generierung von entsprechenden Events zur Verfügung.

Viele bestehende dynamische Analyse-Frameworks bauen auf *TaintDroid* auf (siehe Kapitel 3.2.2). Das Ziel von *TaintDroid* ist es, *privacy leaks* zu erkennen. Auch in dieser Arbeit wäre die Nutzung von *TaintDroid* für die Generierung zusätzlicher Features von Vorteil. Eine Portierung von *TaintDroid* auf die aktuelle Android Version sowie auf die x86-Architektur wäre denkbar.

Auch die Aufzeichnung aller Methoden-Aufrufe während der Laufzeit in einer dafür modifizierten Dalvik-VM wäre denkbar. Die gesammelten Daten könnten, ähnlich wie die während der statischen Analyse generierten Methoden-Signaturen (siehe Listing 4), als Features dienen.

Bessere (dynamische) Feature-Vektoren

Wie in Kapitel 5.4 und besonders in Tabelle 11 zu erkennen ist, konnten erheblich mehr Features aus der statischen als aus der dynamischen Analyse gewonnen werden. Es wird vermutet (siehe Abbildung 29), dass eine zu geringe Anzahl an Features für das schlechte Abschneiden der beiden Datensätze „D50“ und „T50“ verantwortlich war. Aus diesem Grund sollten weitere Wege gefunden werden, aussagekräftige Features in der dynamischen Analyse zu generieren. Diese könnten beispielsweise aus der Nutzung von *TaintDroid*, der Aufzeichnung von Methoden-Aufrufen in der Dalvik-VM oder aus der Analyse der ohnehin bereits gesammelten *Logcat*-Daten stammen. Mit besseren Feature-Vektoren, vor allem aus der dynamischen Analyse, sollte es in weiterer Folge möglich sein, die CP-ANN-Modelle zum Erkennen von Schadsoftware weiter zu optimieren.

Analyse größerer APK-Datensätze verschiedener Applikations-Marktplätze

Wenn die dynamische Analyse weiter optimiert wurde und das Framework so modifiziert, dass auch die dynamische Analyse effizient parallel ausgeführt werden kann, könnten auch größere APK-Datensätze aus dem offiziellen *Google Play Store* oder anderen Applikations-Marktplätzen analysiert werden, um so gegebenenfalls neue, bisher nicht identifizierte Schadsoftware-Familien oder -Varianten aufzudecken.

Abbildungsverzeichnis

Quellcode Verzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADB	Android Debug Bridge
ANN	Artificial Neural Networks
AOSP	Android Open Source Project
AOT	Ahead Of Time
API	Programmierschnittstelle
APK	Android Package
ART	Android Runtime
CPU	Central Processing Unit
CP-ANN	Counterpropagation Artificial Neural Networks
DAC	Discretionary Access Control
DEX	Dalvik Executable Format
FTP	File Transfer Protocol
GC	Garbage Collection
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
IPC	Inter-process Communication
JAR	Java Archive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MAC	Mandatory Access Control
ML	Maschinelles Lernen
NDK	Native Development Kit
OHA	Open Handset Alliance
OVA	Open Virtualization Format
PID	Process Identifier
PUA	potenziell unerwünschte Applikationen
RPC	Remote Procedure Call
SDK	Software Development Kit
SHA-1	Secure Hash Algorithm - Version 1
SOM	Self-Organizing Maps
UID	User Identifier
VM	Virtuelle Maschine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
XML	Extensible Markup Language

Literaturverzeichnis

Alcatel-Lucent. 2014. »Kindsight Security Labs Malware Report – H1 2014«. 9. April. Besucht am 27. September 2014. `http://resources.alcatel-lucent.com/?cid=180437`.

Amalfitano, Domenico, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine und Atif M Memon. 2012. »Using GUI ripping for automated testing of Android applications«. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering,* 258–261. ACM.

Andersen, Birgitte Bo, Lise Korbo und Bente Pakkenberg. 1992. »A quantitative study of the human cerebellum with unbiased stereological techniques«. *Journal of Comparative Neurology* 326 (4): 549–560.

Anderson, Paul. 2008. »The use and limitations of static-analysis tools to improve software quality«. *CrossTalk: The Journal of Defense Software Engineering* 21 (6): 18–21.

App-Market. 2012. »Android Architecture – The Key Concepts of Android OS«. 17. Februar. Besucht am 12. September 2014. `http://www.Android-app-market.com/Android-architecture.html`.

Bach, Maurice J. 1986. *The Design of the UNIX Operating System.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-201799-7.

Ballabio, Davide, Viviana Consonni und Roberto Todeschini. 2009. »The Kohonen and CP-ANN toolbox: a collection of MATLAB modules for self organizing maps and counterpropagation artificial neural networks«. *Chemometrics and Intelligent Laboratory Systems* 98 (2): 115–122.

Barrera, David, H Güneş Kayacik, Paul C van Oorschot und Anil Somayaji. 2010. »A methodology for empirical analysis of permission-based security models and its application to Android«. In *Proceedings of the 17th ACM Conference on Computer and Communications Security,* 73–84. ACM.

Beresford, Alastair R, Andrew Rice, Nicholas Skehin und Ripduman Sohan. 2011. »MockDroid: trading privacy for application functionality on smartphones«. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications,* 49–54. ACM.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 0387310738.

Blasing, Thomas, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe und Sahin Albayrak. 2010. »An Android application sandbox system for suspicious software detection«. In *5th International Conference on Malicious and unwanted software (MALWARE),* 55–62. IEEE.

Bornstein, Dan. 2008. »Dalvik VM Internals«. 3. Juni. Besucht am 14. September 2014. `https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf`.

Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer und Ahmad-Reza Sadeghi. 2011. »Xmandroid: A new Android evolution to mitigate privilege escalation attacks«. *Technische Universität Darmstadt, Technical Report TR-2011-04.*

Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi und Bhargava Shastry. 2012. »Towards Taming Privilege-Escalation Attacks on Android.« In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012.*

Burguera, Iker, Urko Zurutuza und Simin Nadjm-Tehrani. 2011. »Crowdroid: behavior-based malware detection system for Android«. In *Proceedings of the 1st ACM Workshop on Security and privacy in smartphones and mobile devices,* 15–26. ACM.

Cesare, Silvio, und Yang Xiang. 2010. »Classification of malware using structured control flow«. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107,* 61–70. Australian Computer Society, Inc.

Chin, Erika, Adrienne Porter Felt, Kate Greenwood und David Wagner. 2011. »Analyzing inter-application communication in Android«. In *Proceedings of the 9th international Conference on Mobile systems, applications, and services,* 239–252. ACM.

Davi, Lucas, Alexandra Dmitrienko, Ahmad-Reza Sadeghi und Marcel Winandy. 2011. »Privilege escalation attacks on Android«. In *Information Security,* 346–360. Springer.

Desnos, Anthony, und Geoffroy Gueguen. 2011. »Android: From reversing to decompilation«. In *Proc. of Black Hat Abu Dhabi,* 77–101. Black Hat.

Elish, Karim O, Danfeng Yao und Barbara G Ryder. 2012. »User-centric dependence analysis for identifying malicious mobile apps«. In *Workshop on Mobile Security Technologies.*

Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel und Anmol N. Sheth. 2010. »TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones«. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation,* 1–6. OSDI'10. Vancouver, BC, Canada: USENIX Association.

Enck, William, Damien Octeau, Patrick McDaniel und Swarat Chaudhuri. 2011. »A Study of Android Application Security«. In *Proceedings of the 20th USENIX Conference on Security,* 21–21. SEC'11. San Francisco, CA: USENIX Association.

Enck, William, Machigar Ongtang und Patrick McDaniel. 2009. »Understanding Android security«. *Security & Privacy, IEEE* 7 (1): 50–57.

Farmer, Dan, und Wietse Venema. 2004. *Forensic Discovery.* Addison Wesley Professional. ISBN: 020163497X.

Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song und David Wagner. 2011. »Android permissions demystified«. In *Proceedings of the 18th ACM Conference on Computer and communications security,* 627–638. ACM.

Felt, Adrienne Porter, Matthew Finifter, Erika Chin, Steve Hanna und David Wagner. 2011. »A survey of mobile malware in the wild«. In *Proceedings of the 1st ACM Workshop on Security and privacy in smartphones and mobile devices,* 3–14. ACM.

Felt, Adrienne Porter, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin und David Wagner. 2012. »Android Permissions: User Attention, Comprehension, and Behavior«. In *Proceedings of the Eighth Symposium on Usable Privacy and Security,* 3:1–3:14. SOUPS '12. Washington, D.C.: ACM.

Felt, Adrienne Porter, Helen J. Wang, Alexander Moshchuk, Steve Hanna und Erika Chin. 2011. »Permission Re-Delegation: Attacks and Defenses«. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings.*

Frank, I.E., und Roberto Todeschini. 1994. *The Data Analysis Handbook (Data Handling in Science and Technology).* Elsevier Science. ISBN: 9780080868417.

F-Secure. 2014. »Threat Report Q1 2014«. 26. Mai. Besucht am 28. September 2014. `http://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf`.

Gartner. 2013. »Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time«. 14. August. Besucht am 11. März 2014. `http://www.gartner.com/newsroom/id/2573415`.

Google. 2014a. »Android Documentation - Activity Lifecycle«. Besucht am 16. September. `http://developer.Android.com/reference/Android/app/Activity.html`.

———. 2014b. »Android Documentation - Intent«. Besucht am 18. September. `http://developer.Android.com/reference/Android/content/Intent.html`.

———. 2014c. »Android Guide - Intent«. Besucht am 18. September. `http://developer.Android.com/guide/components/intents-filters.html`.

———. 2014d. »Android NDK«. Besucht am 16. September. `https://developer.Android.com/tools/sdk/ndk/index.html`.

———. 2014e. »Android Security Overview«. Besucht am 8. Oktober. `https://source.android.com/devices/tech/security/`.

———. 2014f. »Application Fundamentals«. Besucht am 15. September. `http://developer.Android.com/guide/components/fundamentals.html`.

———. 2014g. »Dalvik Technical Information«. Besucht am 12. September. `http://source.Android.com/devices/tech/dalvik/index.html`.

———. 2014h. »Introducing ART«. Besucht am 12. September. `http://source.Android.com/devices/tech/dalvik/art.html`.

———. 2014i. »Multiple APK Support«. Besucht am 14. September. `http://developer.Android.com/google/play/publishing/multiple-apks.html`.

———. 2014j. »The Android Source Code«. Besucht am 12. September. `https://source.Android.com/source/`.

Google. 2014k. »UI/Application Exerciser Monkey«. Besucht am 21. September. `http://developer.Android.com/tools/help/monkey.html`.

Grace, Michael, Yajin Zhou, Qiang Zhang, Shihong Zou und Xuxian Jiang. 2012. »Riskranker: scalable and accurate zero-day Android malware detection«. In *Proceedings of the 10th International Conference on Mobile systems, applications, and services,* 281–294. ACM.

Jiang, Xuxian. 2012. »An Evaluation of the Application Verification Service in Android 4.2«. 10. Dezember. Besucht am 11. März 2014. `http://www.csc.ncsu.edu/faculty/jiang/appverify/`.

Jiang, Xuxian, und Yajin Zhou. 2013. »A Survey of Android Malware«. In *Android Malware,* 3–20. SpringerBriefs in Computer Science. Springer New York.

Juniper. 2013. »Third Annual Mobile Threats Report«. 12. Juli. Besucht am 13. März 2014. `http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf`.

Karandikar, Radhika. 2010. »Android Application Launch«. 20. April. Besucht am 16. September 2014. `http://multi-core-dump.blogspot.co.at/2010/04/Android-application-launch.html`.

Kohavi, Ron, u. a. 1995. »A study of cross-validation and bootstrap for accuracy estimation and model selection«. In *International Joint Conference On Artificial Intelligence,* 14:1137–1145. 2.

Kohonen, Teuvo. 1990. »The self-organizing map«. *Proceedings of the IEEE* 78 (9): 1464–1480.

Lantz, Patrik. 2014. »DroidBox: alpha release«. Besucht am 21. September. `http://www.honeynet.org/node/744`.

Lee, Sangchul, und Jae Wook Jeon. 2010. »Evaluating performance of Android platform using native C for embedded systems«. In *International Conference on Control Automation and Systems (ICCAS), 2010,* 1160–1163. IEEE.

Lindorfer, Martina. 2012. »Andrubis: A Tool for Analyzing Unknown Android Applications«. 4. Juni. Besucht am 12. September 2014. `http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/`.

Lockheimer, Hiroshi. 2012. »Android and Security«. 2. Februar. Besucht am 18. September 2014. `http://googlemobile.blogspot.co.at/2012/02/Android-and-security.html`.

Meier, R. 2012. *Professional Android 4 Application Development.* Wrox Professional Guides. John Wiley & Sons. ISBN: 9781118102275.

Mitchell, Tom. 1997. *Machine Learning.* New York: McGraw-Hill. ISBN: 0-07-042807-7.

Moser, Andreas, Christopher Kruegel und Engin Kirda. 2007. »Limits of static analysis for malware detection«. In *Twenty-Third Annual Computer Security Applications Conference, 2007. ACSAC 2007.* 421–430. IEEE.

Nauman, Mohammad, Sohail Khan und Xinwen Zhang. 2010. »Apex: extending Android permission model and enforcement with user-defined runtime constraints«. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security,* 328–332. ACM.

Oberheide, J, und C Miller. 2012. »Dissecting the Android bouncer«. *SummerCon2012, New York.* Besucht am 18. September 2014. `https://jon.oberheide.org/files/summercon12-bouncer.pdf`.

Ongtang, Machigar, Stephen McLaughlin, William Enck und Patrick McDaniel. 2012. »Semantically rich application-centric security in Android«. *Security and Communication Networks* 5 (6): 658–673.

Paleari, Roberto, Lorenzo Martignoni, Giampaolo Fresi Roglia und Danilo Bruschi. 2009. »A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators«. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies,* 2–2. WOOT'09. Montreal, Canada: USENIX Association.

PalmSource, Inc. 2005. »OpenBinder Documentation - Binder IPC Mechanism«. 28. Dezember. Besucht am 19. September 2014. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html`.

Popek, Gerald J, und Robert P Goldberg. 1974. »Formal requirements for virtualizable third generation architectures«. *Communications of the ACM* 17 (7): 412–421.

Raffetseder, Thomas, Christopher Kruegel und Engin Kirda. 2007. »Detecting system emulators«. In *Information Security,* 1–18. Springer.

Rastogi, Vaibhav, Yan Chen und William Enck. 2013. »Appsplayground: automatic security analysis of smartphone applications«. In *Proceedings of the third ACM Conference on Data and application security and privacy,* 209–220. ACM.

Reina, Alessandro, Aristide Fattori und Lorenzo Cavallaro. 2013. »A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors«. *EuroSec, April.*

Ritter, Helge, Thomas Martinetz, Klaus Schulten, Daniel Barsky, Marcus Tesch und Ronald Kates. 1992. *Neural computation and self-organizing maps: an introduction.* Addison Wesley Longman Publishing Co., Inc.

Schmidt, A-D, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe und Sahin Albayrak. 2009. »Static analysis of executables for collaborative malware detection on Android«. In *IEEE International Conference on Communications, 2009. ICC'09.* 1–5. IEEE.

Shabtai, Asaf, Yuval Fledel und Yuval Elovici. 2010. »Securing Android-Powered Mobile Devices Using SELinux.« *IEEE Security & Privacy* 8 (3): 36–44.

Shabtai, Asaf, Uri Kanonov, Yuval Elovici, Chanan Glezer und Yael Weiss. 2012. »"Andromaly": a behavioral malware detection framework for Android devices«. *Journal of Intelligent Information Systems* 38 (1): 161–190.

Smalley, Stephen, und Robert Craig. 2013. »Security Enhanced (SE) Android: Bringing Flexible MAC to Android«. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013.*

Sokolova, Marina, Nathalie Japkowicz und Stan Szpakowicz. 2006. »Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation«. In *AI 2006: Advances in Artificial Intelligence,* 1015–1021. Springer.

Spreitzenbarth, Michael, Felix Freiling, Florian Echtler, Thomas Schreck und Johannes Hoffmann. 2013. »Mobile-sandbox: having a deeper look into Android applications«. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing,* 1808–1815. ACM.

Ször, Péter, und Peter Ferrie. 2001. »Hunting for metamorphic«. In *Virus Bulletin Conference,* 123–144.

Tetko, Igor V, David J Livingstone und Alexander I Luik. 1995. »Neural network studies. 1. Comparison of overfitting and overtraining«. *Journal of chemical information and computer sciences* 35 (5): 826–833.

Weichselbaum, Lukas, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen und Christian Platzer. 2014. »Andrubis: Android Malware Under The Magnifying Glass«. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001.*

Whitwam, R. 2014. »Upcoming Google Play Services Update Will Allow Verify Apps Feature To Run Continuously In The Background« (27. Februar). Besucht am 18. September 2014. `http://www.Androidpolice.com/2014/02/27/upcoming-google-play-services-update-will-allow-verify-apps-feature-to-run-continuously-in-the-background/`.

Yan, Lok-Kwong, und Heng Yin. 2012. »DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.« In *USENIX Security Symposium,* 569–584.

Yegnanarayana, B. 2009. *Artificial neural networks.* PHI Learning Pvt. Ltd. ISBN: 8120312538.

Yin, Hujun. 2008. »The self-organizing maps: Background, theories, extensions and applications«. In *Computational intelligence: A compendium,* 715–762. Springer.

You, Ilsun, und Kangbin Yim. 2010. »Malware Obfuscation Techniques: A Brief Survey«. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on,* 297–300. November.

Zdnet. 2012. »Warning: GTA, Super Mario on Google Play are Android malware«. 11. Juli. Besucht am 11. März 2014. `http://www.zdnet.com/warning-gta-super-mario-on-google-play-are-Android-malware-7000000716/`.

Zhou, Wu, Yajin Zhou, Xuxian Jiang und Peng Ning. 2012. »Detecting repackaged smartphone applications in third-party Android marketplaces«. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy,* 317–326. ACM.

Zhou, Yajin, und Xuxian Jiang. 2012. »Dissecting Android malware: Characterization and evolution«. In *IEEE Symposium on Security and Privacy (SP), 2012,* 95–109. IEEE.

Zhou, Yajin, Zhi Wang, Wu Zhou und Xuxian Jiang. 2012. »Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets«. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012.*

Zhou, Yajin, Xinwen Zhang, Xuxian Jiang und Vincent W Freeh. 2011. »Taming information-stealing smartphone applications (on Android)«. In *Trust and Trustworthy Computing,* 93–107. Springer.

Zupan, Jure, Marjana Novič und Itziar Ruisánchez. 1997. »Kohonen and counterpropagation artificial neural networks in analytical chemistry«. *Chemometrics and Intelligent Laboratory Systems* 38 (1): 1–23.

Anhang

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
16	200	29055,2	0,017879	0,98212	0,97574	0,98851	0,98851	0,97574	0,97557	0,98859
16	125	17899,0	0,018277	0,98172	0,9714	0,99204	0,99204	0,9714	0,97143	0,99204
14	50	5626,9	0,019249	0,98075	0,98007	0,98143	0,98143	0,98007	0,9797	0,98177
14	150	16516,0	0,020531	0,97947	0,97574	0,9832	0,9832	0,97574	0,97544	0,98341
16	25	3713,0	0,021345	0,97865	0,9688	0,98851	0,98851	0,9688	0,9688	0,98851
14	125	13819,0	0,022212	0,97779	0,96707	0,98851	0,98851	0,96707	0,96713	0,98849
12	175	14030,5	0,023511	0,97649	0,96447	0,98851	0,98851	0,96447	0,96462	0,98845
12	150	12140,0	0,023564	0,97644	0,96967	0,9832	0,9832	0,96967	0,96949	0,9833
16	75	9265,6	0,023573	0,97643	0,97054	0,98232	0,98232	0,97054	0,97031	0,98246
16	50	7292,0	0,024431	0,97557	0,96794	0,9832	0,9832	0,96794	0,9678	0,98327
10	75	3735,6	0,02528	0,97472	0,96447	0,98497	0,98497	0,96447	0,9645	0,98496
14	25	2868,9	0,02528	0,97472	0,96447	0,98497	0,98497	0,96447	0,9645	0,98496
16	100	14374,0	0,025359	0,97464	0,97227	0,97701	0,97701	0,97227	0,97186	0,97735
16	150	21416,0	0,025394	0,97461	0,97574	0,97347	0,97347	0,97574	0,9752	0,97405
16	175	24471,1	0,025438	0,97456	0,98007	0,96905	0,96905	0,98007	0,97945	0,96998
14	175	19086,3	0,025731	0,97427	0,96534	0,9832	0,9832	0,96534	0,96528	0,98323
12	25	2106,8	0,028816	0,97118	0,96447	0,9779	0,9779	0,96447	0,96425	0,97803
12	50	4133,7	0,030213	0,96979	0,9714	0,96817	0,96817	0,9714	0,97074	0,96889
12	75	5273,7	0,030859	0,96914	0,94801	0,99027	0,99027	0,94801	0,94915	0,99005
10	150	8403,2	0,03207	0,96793	0,98007	0,95579	0,95579	0,98007	0,97917	0,95766
12	100	8143,5	0,032742	0,96726	0,95927	0,97524	0,97524	0,95927	0,95913	0,97533
12	200	16343,4	0,032936	0,96706	0,97834	0,95579	0,95579	0,97834	0,9774	0,95759
8	150	5352,9	0,033857	0,96614	0,93847	0,99381	0,99381	0,93847	0,94059	0,99358
10	100	5654,4	0,035907	0,96409	0,9662	0,96198	0,96198	0,9662	0,96539	0,96287
14	100	11083,0	0,039215	0,96078	0,94367	0,9779	0,9779	0,94367	0,94449	0,97756
10	175	9677,1	0,040718	0,95928	0,96101	0,95756	0,95756	0,96101	0,96011	0,95851
8	100	3605,5	0,042963	0,95704	0,96447	0,9496	0,9496	0,96447	0,96323	0,95128
12	125	10173,0	0,043592	0,95641	0,93934	0,97347	0,97347	0,93934	0,94022	0,97307
10	25	1473,6	0,043918	0,95608	0,9714	0,94076	0,94076	0,9714	0,96992	0,9436
8	175	6127,2	0,044643	0,95536	0,95581	0,95491	0,95491	0,95581	0,95491	0,95581
14	200	22219,9	0,045015	0,95498	0,94887	0,9611	0,9611	0,94887	0,94852	0,96137
14	75	7168,0	0,048233	0,95177	0,96101	0,94253	0,94253	0,96101	0,9595	0,94463

Tabelle 12: F50 Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
10	125	7055,7	0,05284	0,94716	0,93588	0,95844	0,95844	0,93588	0,9361	0,9583
10	200	11380,1	0,056147	0,94385	0,95667	0,93103	0,93103	0,95667	0,95467	0,93401
10	50	2875,5	0,056818	0,94318	0,9792	0,90716	0,90716	0,9792	0,97714	0,91498
8	50	1839,6	0,05904	0,94096	0,89341	0,98851	0,98851	0,89341	0,90089	0,98755
6	100	2098,6	0,06443	0,93557	0,98873	0,8824	0,8824	0,98873	0,98714	0,8956
6	150	3107,0	0,067103	0,9329	0,90381	0,96198	0,96198	0,90381	0,90742	0,96041
8	75	2331,8	0,067202	0,9328	0,87002	0,99558	0,99558	0,87002	0,88245	0,99504
8	125	4500,8	0,067642	0,93236	0,91334	0,95137	0,95137	0,91334	0,91497	0,95041
8	25	940,2	0,068925	0,93108	0,90901	0,95314	0,95314	0,90901	0,91124	0,95191
6	175	3513,4	0,070806	0,92919	0,9636	0,89478	0,89478	0,9636	0,96015	0,90333
8	200	7222,1	0,079101	0,9209	0,90988	0,93192	0,93192	0,90988	0,91019	0,93168
4	125	1165,3	0,080209	0,91979	0,84489	0,99469	0,99469	0,84489	0,86273	0,99388
6	25	549,1	0,084868	0,91513	0,86828	0,96198	0,96198	0,86828	0,87742	0,95885
6	75	1349,5	0,087012	0,91299	0,9922	0,83378	0,83378	0,9922	0,99055	0,85896
6	50	1071,3	0,089664	0,91034	0,9922	0,82847	0,82847	0,9922	0,99049	0,85512
4	175	1551,2	0,099671	0,90033	0,80243	0,99823	0,99823	0,80243	0,83198	0,99784
6	200	4202,2	0,10427	0,89573	0,99393	0,79752	0,79752	0,99393	0,9923	0,83358
6	125	2610,7	0,10478	0,89522	0,82669	0,96375	0,96375	0,82669	0,84496	0,95879
4	75	630,9	0,11399	0,88601	0,99307	0,77896	0,77896	0,99307	0,991	0,82092
4	150	1362,9	0,11886	0,88114	0,99393	0,76835	0,76835	0,99393	0,99201	0,81405
4	25	247,9	0,11887	0,88113	0,9948	0,76746	0,76746	0,9948	0,99314	0,81361
4	100	916,1	0,1193	0,8807	0,99393	0,76746	0,76746	0,99393	0,992	0,81348
4	200	1856,9	0,11974	0,88026	0,99393	0,76658	0,76658	0,99393	0,99199	0,8129
4	50	486,2	0,12019	0,87981	0,9948	0,76481	0,76481	0,9948	0,99311	0,81188
2	100	155,6	0,18789	0,81211	1	0,62423	0,62423	1	1	0,73084
2	25	44,2	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	75	100,9	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	150	224,9	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	175	251,4	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	200	312,3	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	50	79,4	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208
2	125	189,9	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208

Tabelle 13: F50 Datensatz Messwerte - Fortsetzung

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
14	150	15224,0	0,024112	0,97589	0,98007	0,97171	0,97171	0,98007	0,9795	0,97248
16	150	19773,0	0,02444	0,97556	0,9688	0,98232	0,98232	0,9688	0,96861	0,98243
14	50	5141,1	0,024475	0,97553	0,97227	0,97878	0,97878	0,97227	0,97191	0,97906
16	175	23239,0	0,02581	0,97419	0,97314	0,97524	0,97524	0,97314	0,97266	0,97567
16	125	16489,0	0,026703	0,9733	0,974	0,97259	0,97259	0,974	0,97345	0,97316
16	75	9885,1	0,026951	0,97305	0,95494	0,99116	0,99116	0,95494	0,95567	0,99101
14	200	21378,0	0,027455	0,97255	0,96101	0,98408	0,98408	0,96101	0,96114	0,98403
16	100	13205,0	0,027525	0,97247	0,96794	0,97701	0,97701	0,96794	0,9676	0,97725
14	75	7623,3	0,027853	0,97215	0,95667	0,98762	0,98762	0,95667	0,95716	0,98748
14	125	12722,0	0,028693	0,97131	0,95234	0,99027	0,99027	0,95234	0,95319	0,99009
16	200	26418,0	0,030019	0,96998	0,95234	0,98762	0,98762	0,95234	0,95307	0,98742
16	50	6640,0	0,030585	0,96942	0,96447	0,97436	0,97436	0,96447	0,96413	0,97461
16	25	3396,0	0,030743	0,96926	0,98007	0,95844	0,95844	0,98007	0,97922	0,9601
10	125	6546,3	0,031672	0,96833	0,9844	0,95225	0,95225	0,9844	0,98356	0,95462
12	150	11189,0	0,032919	0,96708	0,9766	0,95756	0,95756	0,9766	0,97568	0,95915
14	175	18255,0	0,034572	0,96543	0,96534	0,96552	0,96552	0,96534	0,96466	0,96618
14	100	10210,0	0,035041	0,96496	0,96794	0,96198	0,96198	0,96794	0,96711	0,96293
10	200	10562,9	0,035952	0,96405	0,97054	0,95756	0,95756	0,97054	0,96956	0,9589
12	125	9337,8	0,03596	0,96404	0,9714	0,95668	0,95668	0,9714	0,9704	0,95812
14	25	2628,8	0,036429	0,96357	0,974	0,95314	0,95314	0,974	0,97292	0,95497
12	175	13241,9	0,037287	0,96271	0,9714	0,95402	0,95402	0,9714	0,97032	0,95567
10	50	2614,0	0,039427	0,96057	0,96447	0,95668	0,95668	0,96447	0,96349	0,95783
12	100	7486,3	0,040267	0,95973	0,96014	0,95933	0,95933	0,96014	0,95933	0,96014
12	200	15498,4	0,040328	0,95967	0,9662	0,95314	0,95314	0,9662	0,96509	0,95462
4	50	220,4	0,041116	0,95888	0,95667	0,9611	0,9611	0,95667	0,95602	0,96167
12	25	1931,4	0,042255	0,95774	0,9818	0,93369	0,93369	0,9818	0,9805	0,93791
10	25	1357,6	0,042849	0,95715	0,95321	0,9611	0,9611	0,95321	0,95267	0,96154
10	75	3916,7	0,042875	0,95712	0,95581	0,95844	0,95844	0,95581	0,95507	0,95913
12	75	5577,5	0,042946	0,95705	0,96274	0,95137	0,95137	0,96274	0,96157	0,95283
10	100	5247,9	0,044608	0,95539	0,95234	0,95844	0,95844	0,95234	0,95171	0,95899
10	150	7865,6	0,044829	0,95517	0,974	0,93634	0,93634	0,974	0,97245	0,9398
12	50	3745,2	0,04513	0,95487	0,96014	0,9496	0,9496	0,96014	0,95893	0,95107

Tabelle 14: S50 Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
8	75	2509,7	0,046845	0,95315	0,95494	0,95137	0,95137	0,95494	0,9539	0,95246
10	175	9058,1	0,048541	0,95146	0,99133	0,91158	0,91158	0,99133	0,99039	0,91961
8	25	870,3	0,050206	0,94979	0,93761	0,96198	0,96198	0,93761	0,93793	0,96178
8	50	1667,2	0,051539	0,94846	0,9818	0,91512	0,91512	0,9818	0,98011	0,92189
8	125	4196,4	0,055227	0,94477	0,95321	0,93634	0,93634	0,95321	0,95148	0,93857
8	175	5977,1	0,055458	0,94454	0,93241	0,95668	0,95668	0,93241	0,93276	0,95644
8	150	5038,8	0,061876	0,93812	0,95494	0,92131	0,92131	0,95494	0,95247	0,92527
8	200	6786,9	0,070277	0,92972	0,91161	0,94783	0,94783	0,91161	0,91312	0,94689
6	75	1439,8	0,075942	0,92406	0,99047	0,85765	0,85765	0,99047	0,98879	0,87653
6	125	2447,0	0,078152	0,92185	0,99047	0,85323	0,85323	0,99047	0,98873	0,87319
8	100	3366,5	0,078802	0,9212	0,88042	0,96198	0,96198	0,88042	0,88744	0,9594
6	200	3912,8	0,081689	0,91831	0,99047	0,84615	0,84615	0,99047	0,98864	0,86788
6	150	2926,5	0,083488	0,91651	0,86308	0,96994	0,96994	0,86308	0,8741	0,96699
6	25	506,4	0,088798	0,9112	0,99393	0,82847	0,82847	0,99393	0,99258	0,85533
6	175	3444,1	0,095266	0,90473	0,84749	0,96198	0,96198	0,84749	0,86076	0,95788
6	100	1955,9	0,096773	0,90323	0,99567	0,81079	0,81079	0,99567	0,99458	0,84299
6	50	963,7	0,098095	0,90191	0,86482	0,93899	0,93899	0,86482	0,87192	0,93533
4	100	456,5	0,11621	0,88379	0,99393	0,77365	0,77365	0,99393	0,99206	0,81753
4	25	109,6	0,11708	0,88292	0,99307	0,77277	0,77277	0,99307	0,99093	0,81682
4	200	1422,4	0,11709	0,88291	0,99393	0,77188	0,77188	0,99393	0,99205	0,81637
4	175	1226,6	0,11798	0,88202	0,99393	0,77011	0,77011	0,99393	0,99203	0,81521
4	75	574,7	0,11843	0,88157	0,9948	0,76835	0,76835	0,9948	0,99314	0,81418
4	125	887,6	0,12152	0,87848	0,9948	0,76216	0,76216	0,9948	0,99309	0,81016
4	150	1047,5	0,12955	0,87045	0,95841	0,78249	0,78249	0,95841	0,94855	0,81805
2	75	100,9	0,19584	0,80416	1	0,60831	0,60831	1	1	0,7226
2	200	285,5	0,19584	0,80416	1	0,60831	0,60831	1	1	0,7226
2	125	170,0	0,19629	0,80371	1	0,60743	0,60743	1	1	0,72215
2	25	36,4	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208
2	50	66,2	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208
2	100	137,1	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208
2	150	212,5	0,19761	0,80239	1	0,60477	0,60477	1	1	0,7208
2	175	247,4	0,24757	0,75243	1	0,50486	0,50486	1	1	0,67328

Tabelle 15: S50 Datensatz Messwerte - Fortsetzung

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
16	75	722,5	0,38192	0,61808	0,56066	0,67551	0,67551	0,56066	0,6011	0,63807
16	125	1155,9	0,38244	0,61756	0,56846	0,66667	0,66667	0,56846	0,60224	0,63504
16	100	912,4	0,39031	0,60969	0,55979	0,65959	0,65959	0,55979	0,5949	0,62658
16	200	1917,7	0,39273	0,60727	0,53726	0,67728	0,67728	0,53726	0,58923	0,62944
16	175	1689,0	0,39505	0,60495	0,50433	0,70557	0,70557	0,50433	0,58248	0,63607
16	25	249,0	0,39577	0,60423	0,57539	0,63307	0,63307	0,57539	0,5937	0,61538
16	50	431,2	0,39997	0,60003	0,55373	0,64633	0,64633	0,55373	0,58668	0,61501
16	150	1399,8	0,40065	0,59935	0,5338	0,6649	0,6649	0,5338	0,58295	0,6191
14	125	371,3	0,40344	0,59656	0,59012	0,60301	0,60301	0,59012	0,59048	0,60265
14	50	136,3	0,40531	0,59469	0,55719	0,63218	0,63218	0,55719	0,5832	0,60718
14	100	298,8	0,40566	0,59434	0,54766	0,64103	0,64103	0,54766	0,5814	0,60886
14	200	495,8	0,4066	0,5934	0,50953	0,67728	0,67728	0,50953	0,57508	0,617
14	75	192,9	0,40812	0,59188	0,52946	0,65429	0,65429	0,52946	0,57677	0,60978
14	25	167,5	0,40926	0,59074	0,59792	0,58355	0,58355	0,59792	0,58719	0,59432
14	150	445,0	0,4097	0,5903	0,5104	0,6702	0,6702	0,5104	0,57294	0,61227
14	175	443,7	0,41115	0,58885	0,52253	0,65517	0,65517	0,52253	0,57353	0,60725
12	100	170,7	0,41319	0,58681	0,54939	0,62423	0,62423	0,54939	0,57586	0,59868
12	175	280,6	0,41748	0,58252	0,49307	0,67197	0,67197	0,49307	0,56506	0,60532
12	150	267,3	0,41894	0,58106	0,59272	0,56941	0,56941	0,59272	0,5781	0,58412
12	125	210,9	0,42263	0,57737	0,52166	0,63307	0,63307	0,52166	0,56467	0,59194
12	200	382,2	0,42275	0,57725	0,4896	0,6649	0,6649	0,4896	0,56078	0,59852
12	50	75,6	0,42595	0,57405	0,58666	0,56145	0,56145	0,58666	0,57104	0,57715
12	75	131,3	0,42652	0,57348	0,51213	0,63484	0,63484	0,51213	0,5605	0,58865
12	25	49,3	0,4285	0,5715	0,53293	0,61008	0,61008	0,53293	0,56143	0,58239
10	75	78,7	0,43239	0,56761	0,52426	0,61096	0,61096	0,52426	0,55726	0,57895
10	50	44,1	0,43275	0,56725	0,60312	0,53139	0,53139	0,60312	0,56752	0,5677
10	125	119,3	0,4338	0,5662	0,4896	0,64279	0,64279	0,4896	0,55243	0,58308
10	200	231,1	0,43708	0,56292	0,59445	0,53139	0,53139	0,59445	0,56221	0,56414
10	25	31,0	0,43717	0,56283	0,55979	0,56587	0,56587	0,55979	0,55749	0,56816
10	150	149,3	0,43725	0,56275	0,56759	0,55791	0,55791	0,56759	0,55841	0,5671
10	175	201,6	0,44091	0,55909	0,5364	0,58179	0,58179	0,5364	0,55155	0,56685
10	100	94,6	0,44257	0,55743	0,48267	0,63218	0,63218	0,48267	0,54497	0,57246

Tabelle 16: D50 Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
8	200	129,8	0,44441	0,55559	0,57626	0,53492	0,53492	0,57626	0,55302	0,55835
8	100	58,4	0,44544	0,55456	0,50347	0,60566	0,60566	0,50347	0,54452	0,56573
8	125	72,8	0,44737	0,55263	0,56326	0,542	0,542	0,56326	0,54879	0,55651
8	75	49,8	0,44762	0,55238	0,54419	0,56057	0,56057	0,54419	0,54655	0,55822
8	175	115,1	0,45268	0,54732	0,60659	0,48806	0,48806	0,60659	0,54871	0,5473
8	50	27,6	0,45304	0,54696	0,55546	0,53846	0,53846	0,55546	0,54278	0,55116
8	25	17,3	0,45337	0,54663	0,58752	0,50575	0,50575	0,58752	0,5458	0,5481
8	150	91,3	0,45369	0,54631	0,53293	0,55968	0,55968	0,53293	0,5401	0,55256
6	150	57,3	0,46005	0,53995	0,59272	0,48718	0,48718	0,59272	0,53967	0,54114
6	200	79,8	0,46109	0,53891	0,5208	0,55703	0,55703	0,5208	0,53254	0,54537
6	50	18,4	0,46233	0,53767	0,59965	0,47569	0,47569	0,59965	0,538	0,53852
6	100	35,8	0,46256	0,53744	0,66551	0,40937	0,40937	0,66551	0,54535	0,53482
6	25	11,1	0,46324	0,53676	0,55893	0,51459	0,51459	0,55893	0,53346	0,5402
6	175	74,1	0,465	0,535	0,55806	0,51194	0,51194	0,55806	0,53168	0,53846
6	125	45,8	0,46542	0,53458	0,55546	0,5137	0,5137	0,55546	0,53108	0,5382
6	75	31,6	0,46884	0,53116	0,58752	0,4748	0,4748	0,58752	0,53011	0,53302
4	125	27,2	0,47027	0,52973	0,68458	0,37489	0,37489	0,68458	0,53807	0,52772
4	25	6,3	0,47078	0,52922	0,474	0,58444	0,58444	0,474	0,52129	0,53786
4	75	18,5	0,47091	0,52909	0,70451	0,35367	0,35367	0,70451	0,53981	0,52655
4	150	34,1	0,47151	0,52849	0,67678	0,38019	0,38019	0,67678	0,53549	0,52699
4	175	42,7	0,47399	0,52601	0,65945	0,39257	0,39257	0,65945	0,53047	0,52555
4	200	49,6	0,47459	0,52541	0,71837	0,33245	0,33245	0,71837	0,53638	0,52336
4	100	22,4	0,47476	0,52524	0,73484	0,31565	0,31565	0,73484	0,53846	0,52281
4	50	11,2	0,47568	0,52432	0,69497	0,35367	0,35367	0,69497	0,53191	0,52316
2	175	24,6	0,48545	0,51455	0,78596	0,24315	0,24315	0,78596	0,52682	0,51446
2	75	10,6	0,48545	0,51455	0,78596	0,24315	0,24315	0,78596	0,52682	0,51446
2	50	6,9	0,48645	0,51355	0,18891	0,8382	0,8382	0,18891	0,50318	0,54364
2	25	3,5	0,48698	0,51302	0,19757	0,82847	0,82847	0,19757	0,50295	0,54028
2	100	13,7	0,4924	0,5076	0,29636	0,71883	0,71883	0,29636	0,50031	0,51818
2	150	20,7	0,49342	0,50658	0,35269	0,66048	0,66048	0,35269	0,5	0,51454
2	125	16,9	0,5	0,5	1	0	0	1	0	0,50503
2	200	26,2	0,5	0,5	1	0	0	1	0	0,50503

Tabelle 17: D50 Datensatz Messwerte - Fortsetzung

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
16	200	148,0	0,39047	0,60953	0,66291	0,55615	0,55615	0,66291	0,61788	0,60379
16	125	79,8	0,39162	0,60838	0,64558	0,57118	0,57118	0,64558	0,61232	0,60569
16	150	97,5	0,39176	0,60824	0,61612	0,60035	0,60035	0,61612	0,60517	0,61135
16	175	124,1	0,39264	0,60736	0,65858	0,55615	0,55615	0,65858	0,61486	0,60222
16	75	56,8	0,39669	0,60331	0,62218	0,58444	0,58444	0,62218	0,60255	0,60438
16	25	19,5	0,39742	0,60258	0,60745	0,5977	0,5977	0,60745	0,59876	0,6064
16	100	63,4	0,39861	0,60139	0,59359	0,6092	0,6092	0,59359	0,59499	0,60781
14	200	104,7	0,40082	0,59918	0,68024	0,51813	0,51813	0,68024	0,61361	0,59023
16	50	31,5	0,40116	0,59884	0,67071	0,52697	0,52697	0,67071	0,61066	0,59129
14	25	13,8	0,40572	0,59428	0,64125	0,5473	0,5473	0,64125	0,59923	0,59105
14	75	39,2	0,40872	0,59128	0,63172	0,55084	0,55084	0,63172	0,59447	0,58933
14	175	91,8	0,40911	0,59089	0,62652	0,55526	0,55526	0,62652	0,59301	0,58972
12	75	32,4	0,41061	0,58939	0,68718	0,4916	0,4916	0,68718	0,60632	0,57968
14	150	76,8	0,41228	0,58772	0,54679	0,62865	0,62865	0,54679	0,57618	0,60038
14	125	61,2	0,41292	0,58708	0,61005	0,5641	0,5641	0,61005	0,5864	0,58814
14	50	24,3	0,41298	0,58702	0,57192	0,60212	0,60212	0,57192	0,57957	0,59459
14	100	47,4	0,41587	0,58413	0,63865	0,52962	0,52962	0,63865	0,58957	0,58077
12	25	11,0	0,41612	0,58388	0,62045	0,5473	0,5473	0,62045	0,58562	0,58306
12	100	36,8	0,41884	0,58116	0,67071	0,4916	0,4916	0,67071	0,59402	0,57376
12	200	83,9	0,41898	0,58102	0,64038	0,52166	0,52166	0,64038	0,58706	0,57734
12	150	56,4	0,42155	0,57845	0,63258	0,52431	0,52431	0,63258	0,58309	0,57571
12	125	45,6	0,42266	0,57734	0,65511	0,49956	0,49956	0,65511	0,58671	0,57186
12	50	18,9	0,42403	0,57597	0,65858	0,49337	0,49337	0,65858	0,58613	0,57014
10	25	8,3	0,42732	0,57268	0,63432	0,51105	0,51105	0,63432	0,578	0,56965
12	175	73,3	0,42771	0,57229	0,54246	0,60212	0,60212	0,54246	0,56328	0,58178
10	50	14,1	0,42949	0,57051	0,63085	0,51017	0,51017	0,63085	0,57527	0,56786
10	75	24,3	0,42992	0,57008	0,67331	0,46684	0,46684	0,67331	0,58343	0,56304
10	100	27,9	0,43005	0,56995	0,68544	0,45447	0,45447	0,68544	0,58609	0,56179
10	150	41,7	0,43085	0,56915	0,63432	0,50398	0,50398	0,63432	0,5746	0,56613
10	125	34,8	0,43385	0,56615	0,66811	0,46419	0,46419	0,66811	0,57819	0,55991
10	175	54,9	0,43804	0,56196	0,55806	0,56587	0,56587	0,55806	0,55652	0,5674
8	100	20,5	0,4382	0,5618	0,74783	0,37577	0,37577	0,74783	0,59358	0,55003

Tabelle 18: T50 Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
8	75	18,0	0,44076	0,55924	0,65251	0,46596	0,46596	0,65251	0,56789	0,5549
8	150	31,5	0,44407	0,55593	0,62998	0,48187	0,48187	0,62998	0,5607	0,55369
8	200	46,9	0,44485	0,55515	0,66291	0,44739	0,44739	0,66291	0,56536	0,55036
8	125	26,1	0,4449	0,5551	0,66811	0,44209	0,44209	0,66811	0,56625	0,54993
8	50	10,7	0,44595	0,55405	0,59705	0,51105	0,51105	0,59705	0,55417	0,55475
10	200	63,5	0,4466	0,5534	0,66118	0,44562	0,44562	0,66118	0,56313	0,54892
8	175	42,8	0,44797	0,55203	0,57886	0,5252	0,5252	0,57886	0,55	0,55436
8	25	6,3	0,44933	0,55067	0,62565	0,47569	0,47569	0,62565	0,55464	0,54905
6	50	7,6	0,45224	0,54776	0,65078	0,44474	0,44474	0,65078	0,55519	0,5446
6	150	22,7	0,45441	0,54559	0,68977	0,40141	0,40141	0,68977	0,55911	0,54039
6	75	13,0	0,45642	0,54358	0,67071	0,41645	0,41645	0,67071	0,55347	0,53975
6	125	19,1	0,45662	0,54338	0,68977	0,39699	0,39699	0,68977	0,55638	0,53857
6	175	31,4	0,45762	0,54238	0,65771	0,42706	0,42706	0,65771	0,55011	0,53945
6	100	15,7	0,45875	0,54125	0,72617	0,35632	0,35632	0,72617	0,5605	0,53512
6	25	4,8	0,4595	0,5405	0,66898	0,41202	0,41202	0,66898	0,54953	0,53723
6	200	33,9	0,46065	0,53935	0,7383	0,34041	0,34041	0,7383	0,56041	0,53317
4	25	3,0	0,46262	0,53738	0,58492	0,48983	0,48983	0,58492	0,5363	0,53914
4	125	13,0	0,46609	0,53391	0,57799	0,48983	0,48983	0,57799	0,53218	0,53617
4	100	10,5	0,46813	0,53187	0,64818	0,41556	0,41556	0,64818	0,53653	0,53087
4	175	20,0	0,47046	0,52954	0,66031	0,39876	0,39876	0,66031	0,53499	0,52843
4	200	22,6	0,47111	0,52889	0,63692	0,42087	0,42087	0,63692	0,53184	0,52878
4	50	5,4	0,47402	0,52598	0,57539	0,47657	0,47657	0,57539	0,52381	0,52866
4	150	14,9	0,47536	0,52464	0,57626	0,47303	0,47303	0,57626	0,52246	0,52736
4	75	8,5	0,47708	0,52292	0,70277	0,34306	0,34306	0,70277	0,53078	0,52188
2	200	14,0	0,47793	0,52207	0,65598	0,38815	0,38815	0,65598	0,52512	0,52243
2	150	11,6	0,47797	0,52203	0,65945	0,38462	0,38462	0,65945	0,52536	0,52231
2	75	5,6	0,48053	0,51947	0,65078	0,38815	0,38815	0,65078	0,52138	0,52044
2	25	1,9	0,48131	0,51869	0,77036	0,26702	0,26702	0,77036	0,53263	0,51746
2	50	4,3	0,48216	0,51784	0,7669	0,26879	0,26879	0,7669	0,53054	0,51694
2	100	7,8	0,48265	0,51735	0,7721	0,2626	0,2626	0,7721	0,53036	0,51652
2	125	9,8	0,48277	0,51723	0,78336	0,25111	0,25111	0,78336	0,53184	0,51628
2	175	12,9	0,48526	0,51474	0,89861	0,13086	0,13086	0,89861	0,55849	0,51337

Tabelle 19: T50 Datensatz Messwerte - Fortsetzung

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
14	100	13495,0	0,019223	0,98078	0,97747	0,98408	0,98408	0,97747	0,97717	0,98429
14	200	27766,0	0,02131	0,97869	0,96534	0,99204	0,99204	0,96534	0,96558	0,99199
16	75	12185,0	0,021761	0,97824	0,9662	0,99027	0,99027	0,9662	0,96635	0,99023
14	75	9414,4	0,022229	0,97777	0,9688	0,98674	0,98674	0,9688	0,96875	0,98676
16	150	26171,0	0,022618	0,97738	0,9636	0,99116	0,99116	0,9636	0,96389	0,99109
14	175	23399,6	0,023538	0,97646	0,96707	0,98585	0,98585	0,96707	0,96704	0,98587
16	200	35199,6	0,024077	0,97592	0,9766	0,97524	0,97524	0,9766	0,97611	0,97576
16	50	8910,1	0,024855	0,97514	0,9662	0,98408	0,98408	0,9662	0,96615	0,98411
16	100	17491,0	0,025625	0,97438	0,95494	0,99381	0,99381	0,95494	0,95578	0,99369
14	25	3490,3	0,027145	0,97285	0,974	0,97171	0,97171	0,974	0,97343	0,97232
14	150	20191,0	0,027534	0,97247	0,9688	0,97613	0,97613	0,9688	0,96842	0,97642
16	175	30242,0	0,028029	0,97197	0,974	0,96994	0,96994	0,974	0,97338	0,97064
16	125	21876,0	0,032388	0,96761	0,96794	0,96729	0,96729	0,96794	0,96729	0,96794
14	50	6815,5	0,033167	0,96683	0,95754	0,97613	0,97613	0,95754	0,9575	0,97615
16	25	4508,3	0,035272	0,96473	0,94714	0,98232	0,98232	0,94714	0,94795	0,98203
14	125	16896,0	0,036031	0,96397	0,97834	0,9496	0,9496	0,97834	0,97725	0,95194

Tabelle 20: F30 Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
16	50	2928,3	0,073328	0,92667	0,90728	0,94607	0,94607	0,90728	0,90909	0,94495
16	125	7159,6	0,076308	0,92369	0,89601	0,95137	0,95137	0,89601	0,89967	0,94949
16	25	1489,1	0,076706	0,92329	0,89168	0,95491	0,95491	0,89168	0,89627	0,95278
14	25	1123,4	0,078147	0,92185	0,90295	0,94076	0,94076	0,90295	0,90476	0,93959
16	175	9955,4	0,07843	0,92157	0,88735	0,95579	0,95579	0,88735	0,89265	0,95345
16	100	5726,4	0,078863	0,92114	0,88648	0,95579	0,95579	0,88648	0,89191	0,9534
16	75	4046,4	0,078881	0,92112	0,88821	0,95402	0,95402	0,88821	0,89321	0,95172
16	200	11500,0	0,079908	0,92009	0,85875	0,98143	0,98143	0,85875	0,87196	0,97925
14	125	5558,8	0,079941	0,92006	0,90555	0,93457	0,93457	0,90555	0,90652	0,93387
14	100	4437,5	0,08003	0,91997	0,91421	0,92573	0,92573	0,91421	0,91361	0,92625
14	75	3116,1	0,081622	0,91838	0,89688	0,93988	0,93988	0,89688	0,89932	0,93835
16	150	8586,4	0,081736	0,91826	0,90815	0,92838	0,92838	0,90815	0,9083	0,92826
14	200	8856,3	0,083726	0,91627	0,88648	0,94607	0,94607	0,88648	0,89092	0,94373
14	150	6623,5	0,084063	0,91594	0,87608	0,95579	0,95579	0,87608	0,88317	0,95287
14	175	7708,1	0,085548	0,91445	0,89168	0,93722	0,93722	0,89168	0,89451	0,93545
14	50	2237,1	0,08569	0,91431	0,86222	0,9664	0,9664	0,86222	0,873	0,96321

Tabelle 21: DT5P Datensatz Messwerte

nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malware)	sensitivity(Benign)	sensitivity(Malware)	precision(Benign)	precision(Malware)
16	25	2946,9	0,019152	0,98085	0,97054	0,99116	0,99116	0,97054	0,97056	0,99115
16	50	5775,5	0,021389	0,97861	0,97314	0,98408	0,98408	0,97314	0,9729	0,98422
16	100	11337,0	0,021761	0,97824	0,9662	0,99027	0,99027	0,9662	0,96635	0,99023
16	125	14138,0	0,021787	0,97821	0,9688	0,98762	0,98762	0,9688	0,96878	0,98763
14	200	17209,2	0,021849	0,97815	0,97487	0,98143	0,98143	0,97487	0,97454	0,98168
14	150	13095,0	0,022194	0,97781	0,96534	0,99027	0,99027	0,96534	0,96552	0,99022
14	100	8755,0	0,023175	0,97682	0,97487	0,97878	0,97878	0,97487	0,97447	0,97911
16	75	7887,1	0,023202	0,9768	0,97747	0,97613	0,97613	0,97747	0,97699	0,97662
14	175	15428,2	0,023485	0,97652	0,96187	0,99116	0,99116	0,96187	0,96223	0,99107
16	175	19973,5	0,024448	0,97555	0,96967	0,98143	0,98143	0,96967	0,96943	0,98158
16	200	22327,5	0,025651	0,97435	0,95754	0,99116	0,99116	0,95754	0,95812	0,99103
16	150	16920,0	0,026641	0,97336	0,96794	0,97878	0,97878	0,96794	0,96766	0,97897
14	125	10929,0	0,027765	0,97224	0,94801	0,99646	0,99646	0,94801	0,94945	0,99636
14	25	2263,4	0,03024	0,96976	0,974	0,96552	0,96552	0,974	0,97326	0,96647
14	50	4419,8	0,034122	0,96588	0,96447	0,96729	0,96729	0,96447	0,96388	0,96783
14	75	6107,1	0,034616	0,96538	0,96967	0,9611	0,9611	0,96967	0,96881	0,96217

Tabelle 22: F5P Datensatz Messwerte

dataset	nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malw.)	sensitivity(Benign)	sensitivity(Malw.)	precision(Benign)	precision(Malw.)
F50	16	200	29055,2	0,017879	0,98212	0,97574	0,98851	0,98851	0,97574	0,97557	0,98859
F50	16	125	17899,0	0,018277	0,98172	0,9714	0,99204	0,99204	0,9714	0,97143	0,99204
F5P	16	25	2946,9	0,019152	0,98085	0,97054	0,99116	0,99116	0,97054	0,97056	0,99115
F50	14	100	13495,0	0,019223	0,98078	0,97747	0,98408	0,98408	0,97747	0,97717	0,98429
F50	14	50	5626,9	0,019249	0,98075	0,98007	0,98143	0,98143	0,98007	0,9797	0,98177
F50	14	150	16516,0	0,020531	0,97947	0,97574	0,9832	0,9832	0,97574	0,97544	0,98341
F5P	14	200	27766,0	0,02131	0,97869	0,96534	0,99204	0,99204	0,96534	0,96558	0,99199
F50	16	25	3713,0	0,021345	0,97865	0,9688	0,98851	0,98851	0,9688	0,9688	0,98851
F5P	16	50	5775,5	0,021389	0,97861	0,97314	0,98408	0,98408	0,97314	0,9729	0,98422
F5P	16	100	11337,0	0,021761	0,97824	0,9662	0,99027	0,99027	0,9662	0,96635	0,99023
F5P	16	75	12185,0	0,021761	0,97824	0,9662	0,99027	0,99027	0,9662	0,96635	0,99023
F50	16	125	14138,0	0,021787	0,97821	0,9688	0,98762	0,98762	0,9688	0,96878	0,98763
F5P	14	200	17209,2	0,021849	0,97815	0,97487	0,98143	0,98143	0,97487	0,97454	0,98168
F5P	14	150	13095,0	0,022194	0,97781	0,96534	0,99027	0,99027	0,96534	0,96552	0,99022
F5P	14	125	13819,0	0,022212	0,97779	0,96707	0,98851	0,98851	0,96707	0,96713	0,98849
F50	14	75	9414,4	0,022229	0,97777	0,9688	0,98674	0,98674	0,9688	0,96875	0,98676
F50	16	150	26171,0	0,022618	0,97738	0,9636	0,99116	0,99116	0,9636	0,96389	0,99109
F50	14	100	8755,0	0,023175	0,97682	0,97487	0,97878	0,97878	0,97487	0,97447	0,97911
S50	16	75	7887,1	0,023202	0,9768	0,97747	0,97613	0,97613	0,97747	0,97699	0,97662
F50	14	175	15428,2	0,023485	0,97652	0,96187	0,99116	0,99116	0,96187	0,96223	0,99107
S50	12	175	14030,5	0,023511	0,97649	0,96447	0,98851	0,98851	0,96447	0,96462	0,98845
F5P	14	175	23399,6	0,023538	0,97646	0,96707	0,98585	0,98585	0,96707	0,96704	0,98587
S50	12	150	12140,0	0,023564	0,97644	0,96967	0,9832	0,9832	0,96967	0,96949	0,9833
F50	16	75	9265,6	0,023573	0,97643	0,97054	0,98232	0,98232	0,97054	0,97031	0,98246
F50	16	200	35199,6	0,024077	0,97592	0,9766	0,97524	0,97524	0,9766	0,97611	0,97576
F50	14	150	15224,0	0,024112	0,97589	0,98007	0,97171	0,97171	0,98007	0,9795	0,97248
F50	16	50	7292,0	0,024431	0,97557	0,96794	0,9832	0,9832	0,96794	0,9678	0,98327
F50	16	150	19773,0	0,02444	0,97556	0,9688	0,98232	0,98232	0,9688	0,96861	0,98243
F5P	16	175	19973,5	0,024448	0,97555	0,96967	0,98143	0,98143	0,96967	0,96943	0,98158
F50	14	50	5141,1	0,024475	0,97553	0,97227	0,97878	0,97878	0,97227	0,97191	0,97906
S50	16	50	8910,1	0,024855	0,97514	0,9662	0,98408	0,98408	0,9662	0,96615	0,98411
F5P	10	75	3735,6	0,02528	0,97472	0,96447	0,98497	0,98497	0,96447	0,9645	0,98496

Tabelle 23: Top-10 beziehungsweise Top-32 CP-ANN-Modelle

dataset	nsize	epochs	elapsed-time	error-rate	accuracy	specificity(Benign)	specificity(Malw.)	sensitivity(Benign)	sensitivity(Malw.)	precision(Benign)	precision(Malw.)
F50 16x200	16	200	21513,6	0,028392	0,971608	0,966205	0,977011	0,977011	0,966205	0,965909	0,977213
F5P 16x5	16	50	4360,2	0,028808	0,971192	0,963605	0,97878	0,97878	0,963605	0,963446	0,978873
F50 14x150	14	150	12470,8	0,02925	0,97075	0,963605	0,977896	0,977896	0,963605	0,963415	0,978012
F5P 16x100	16	100	8583,8	0,029665	0,970335	0,961005	0,979664	0,979664	0,961005	0,960971	0,979682
F5P 16x25	16	25	2239,4	0,031522	0,968478	0,969671	0,967286	0,967286	0,969671	0,968999	0,967993
F50 16x25	16	25	2839,5	0,031451	0,968549	0,962738	0,974359	0,974359	0,962738	0,962445	0,974561
F50 14x50	14	50	4254,0	0,032327	0,967673	0,961872	0,973475	0,973475	0,961872	0,961572	0,973684
F50 14x100	14	100	8359,9	0,034944	0,965056	0,958406	0,971706	0,971706	0,958406	0,958152	0,97188
F50 16x125	16	125	13506,7	0,035828	0,964172	0,958406	0,969938	0,969938	0,958406	0,958079	0,970175
F5P 14x200	14	200	13152,6	0,035925	0,964075	0,967938	0,960212	0,960212	0,967938	0,967053	0,961274

Tabelle 24: Top-10 CP-ANN-Modelle, vierfache Kreuz-Validierung